Final Report: Intent Specifications

Nancy G. Leveson

We have been investigating the implications of using abstractions based on intent rather than the aggregation and information-hiding abstractions commonly used in software engineering: Cognitive psychologists have shown that intent abstraction is consistent with human problem-solving processes. We believe that new types of specifications and designs based on this concept can assist in understanding and specifying requirements, capturing the most important design rationale information in an efficient and economical way, and supporting the process of identifying and analyzing required changes to minimize the introduction of errors.

The goal of hierarchical abstraction is to allow both top-down and bottom-up reasoning about a complex system. In computer science, we have made much use of (1) part-whole abstractions where each level of a hierarchy represents an aggregation of the components at a lower level and of (2) information-hiding abstractions where each level contains the same conceptual information but hides some details about the concepts, that is, each level is a refinement of the information at a higher level. Each level of our software specifications can be thought of as providing "what" information while the next lower level describes "how." Such hierarchies, however, do not provide information about "why." Higher-level emergent information about purpose or intent cannot be inferred from what we normally include in such specifications. Design errors may result when we either guess incorrectly about higher-level intent or omit it from our decision-making process. For example, while specifying the system requirements for TCAS using a pseudocode specification, we learned (orally from a reviewer) that crossing maneuvers were avoided in the design for safety reasons. This important safety constraint was not represented in the pseudocode (and could not have been unless it had been added in textual comments somewhere).

Each level of an intent abstraction (or what Rasmussen calls a means-end abstraction [Ras86]) contains the goals or purpose for the level below and describes the system in terms of a different set of attributes or language. Higher level goals are not constructed by integrating information from lower levels; instead each level provides different, emergent information with respect to the lower levels. A change of level represents both a shift in concepts and structure for the representation (and not just a refinement of them) as well as a change in the type of information used to characterize the state of the system at that level. Mappings between levels are many-to-many: Components of the lower levels can serve

several purposes while purposes at a higher level may be realized using several components of the lower-level model. These goal-oriented links between levels can be followed in either direction. Changes at higher levels will propagate downward, i.e., require changes in lower levels while design errors at lower levels can only be explained through upward mappings (that is, in terms of the goals the design is expected to achieve).

Consideration of purpose or reason (top-down analysis in an intent hierarchy) has been shown to play a major role in understanding the operation of complex systems [Ras85]. Experts and successful problem solvers tend to focus first on analyzing the functional structure of the problem at a high level of abstraction and then narrow their search for a solution by focusing on more concrete details [GC88]. Representations that constrain search in a way that is explicitly related to the purpose or intent for which the system is designed have been shown to be more effective than those that do not because they facilitate the type of goal-directed behavior exhibited by experts [VCP95]. Therefore, we should be able to improve the problem solving required in software evolution tasks by providing a representation (i.e., specification) of the system that facilitates goal-oriented search by making explicit the goals related to each component.

As stated earlier, purpose or intent abstraction has been used successfully in cognitive engineering by Rasmussen and Vicente for the design of operator interfaces [DV96, Vic91]. We have been applying it to designing system and software specifications. Changes are required (mostly augmentations) to adopt the approach to a different problem. The changes include adding models of the environment and of the human components (operators) in the system along with any displays and controls that operators may use. This change allows the integration of software and human-interface design.

We have also changed the content of the levels of the intent abstraction. System and software specifications of the type we are proposing are organized along two dimensions: intent abstraction and part-whole abstraction. These two dimensions constitute the problem space in which the human navigates. The vertical dimension specifies the level of intent at which the problem is being considered, i.e., the language or model that is currently being used. The part-whole (horizontal) dimension allows users to change their focus of attention to more or less detailed views within each level or model. The information at each level is fully linked to related information at the levels above and below it.

1 Results

We have developed the theory underlying intent specifications by investigating issues concerning the content and organization of the levels and traceability (particularly traceability of safety information). A paper on this topic has been accepted for IEEE Transactions on Software Engineering and is included with the final report. The paper is scheduled to appear in the January 2000 issue of the journal.

We have also investigated the feasbility of writing intent specifications by writing them for industrial problems, the most complex of which has been TCAS II. TCAS II is an on-board collision avoidance system that is required on most commercial aircraft flown in U.S. airspace. The system has been described as one of the most complex automated systems that has been integrated into the avionics of commercial aircraft.

We also used the lessons learned from writing the FAA TCAS II system specification in a language we call RSML. RSML was developed for this application. After many years of use, we have identified important needed improvements and have designed a new language called SpecTRM-RL. SpecTRM is a set of development tools for embedded systems and SpecTRM-RL is the requirements language used to build system models. With respect to intent specifications, SpecTRM-RL is used to specify the third level (blackbox system behavior).

Like RSML, SpecTRM-RL is based on a state machine model. Readability and reviewability with little training was a primary goal for both languages. We believe that readability is enhanced in the preliminary design of SpecTRM-RL. In addition, we have removed the most error-prone feature of RSML (and related state-machine languages, such as Statecharts) which is internally broadcast events. Also, we have added the ability to organize the specification using modes, an abstraction commonly used by engineers to design complex systems. A paper on the design of SpecTRM-RL has been accepted for the IEEE/ACM Conference on Foundations of Software Engineering and will be presented in September 1999. A copy of this paper is also included with this final report

2 Bibliography

[Che81] P. Checkland. *Systems Thinking, Systems Practice.* John Wiley & Sons, 1981.

[DV96] N. Dinadis and K.J. Vicente. Ecological interface design for a power plant feedwater subsystem. *IEEE Transactions on Nuclear Science*, in press.

[GC88] R. Glaser and M. T. H. Chi. Overview. In R. Glaser, M. T. H. Chi, and M. J. Farr, editors, *The Nature of Expertise.* Erlbaum, Hillsdale, New Jersey, 1988.

[Lev95] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, 1995.

[Ras85] J. Rasmussen. The Role of hierarchical knowledge representation in decision making and wystem management. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, no. 2, March/April 1985.

[Ras86] J. Rasmussen. *Information Processing and Human–Machine Interaction: An Approach to Cognitive Engineering.* North Holland, 1986.

[Vic91] K.J. Vicente. Supporting knowledge-based behavior through ecological interface design. Ph.D. Dissertation, University of Illinois at Urbana-Champagne, 1991.

[VCP95] Kim J. Vicente and Klaus Christoffersen and Alex Pereklit. Supporting operator

problem solving through ecological interface design. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(4):529–545, 1995.

[VR92] K.J. Vicente and J. Rasmussen. Ecological interface design: Theoretical foundations. *IEEE Transactions on Systems, Man, and Cybernetics*, vol 22, No. 4, July/August 1992.

To be presented at "Foundations of Software Engineering", Toulouse France, Sept. 1999

(DRAFT)

Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future

Nancy G. Leveson[1], Mats P.E. Heimdahl[2], and Jon Damon Reese

[1] Aeronautics and Astronautics Dept.
MIT
Room 33-406, 77 Massachusetts Ave.
Cambridge, MA 02139-4307
leveson@mit.edu

[2] Computer Science and Engineering Department,
University of Minnesota,
4-192 EE/CS Building, 200 Union Street S.E.
Minneapolis, MN 55455,
heimdahl@cs.umn.edu

Abstract. Previously, we defined a blackbox formal system modeling language called RSML (Requirements State Machine Language). The language was developed over several years while specifying the system requirements for a collision avoidance system for commercial passenger aircraft. During the language development, we received continual feedback and evaluation by FAA employees and industry representatives, which helped us to produce a specification language that is easily learned and used by application experts.

Since the completion of the RSML project, we have continued our research on specification languages. This research is part of a larger effort to investigate the more general problem of providing tools to assist in developing embedded systems. Our latest experimental toolset is called SpecTRM (Specification Tools and Requirements Methodology), and the formal specification language is SpecTRM-RL (SpecTRM Requirements Language).

This paper describes what we have learned from our use of RSML and how those lessons were applied to the design of SpecTRM-RL. We discuss our goals for SpecTRM-RL and the design features that support each of these goals.

1 Introduction

In 1994, we published a paper describing a blackbox formal system modeling language called RSML (Requirements State Machine Language). The language was developed over several years during an effort to specify the system requirements for a collision avoidance system for commercial passenger aircraft called TCAS II (Traffic Alert and Collision Avoidance System). Because this was to be the

This work was partially supported by NAG-1-2020.

official FAA (Federal Aviation Administration) specification, it was developed with continual feedback and evaluation by FAA employees, airframe manufacturers, airline representatives, pilots, and other external reviewers. Most of the reviewers were not software engineers or even computer scientists, and we believe this helped in producing a specification language that is easily learned and used by application experts. RSML is still being used by the FAA, its subcontractors, and RTCA committees to specify the upgrades and changes to TCAS II.

Those designing specification languages often have themselves in mind as potential users. However, our familiarity with certain notations, especially mathematical notations, such as predicate calculus, hides their weaknesses. Our first attempts at designing RSML, therefore, were failures: Our notation was clear to us but not to the representatives from the airframe manufacturers, component subcontractors, airlines, and pilot groups that reviewed the TCAS specification during its development. The feedback from a diverse group of users helped us to evaluate the evolving specification language more objectively in terms of what did and did not need to be in the language; how to satisfy our language design criteria; and its practicality, feasibility, and usability.

Due to pressure to meet FAA deadlines for getting TCAS II on aircraft, we were unable to use immediately all the lessons learned from that experience and apply it to the design of RSML. Since that time, we have specified several additional systems including a robot, flight management system, and air traffic control components, each time learning more lessons about the design of formal specification languages. Our research goal is to determine how specification languages can be designed to reflect these lessons. Our research paradigm is to determine important goals for specification languages from experience with industrial applications, to generate hypotheses about how these goals might be accomplished, and then to instantiate these hypotheses in the design of a specification language that we will use in future experimentation. In this way, we hope to build knowledge incrementally about how to most effectively design specification languages.

Our specification language research is part of a larger research effort to investigate the more general problem of providing tools to assist in developing embedded systems. Our latest experimental toolset is called SpecTRM (Specification Tools and Requirements Methodology), and the formal specification language is SpecTRM-RL (SpecTRM Requirements Language). In addition to the general goals we had for designing RSML [9], the lessons we have learned to date have focused our latest efforts on solving the following problems:

1. Through the use of RSML, we have determined that readability and reviewability by domain experts can be further enhanced by minimizing the semantic distance between the reviewer's mental model and the constructed models. The problem we are now addressing is how to construct a modeling language that will allow and encourage modelers to reduce this semantic distance in the models they build.
2. Specifiers are used to including internal design in their specifications and seem to have difficulty building pure blackbox requirements models. So a

second goal was to provide more support and guidance in building software requirements (versus software design) models.

3. We found certain common features of formal specification languages were very error-prone in use. In particular, the use of internal broadcast events accounted for most of the errors found by reviewers of the TCAS specification and also contributed substantially to the difficulty reviewers had in reading the models. A third goal for SpecTRM-RL was to determine if such internal events can be effectively eliminated from state-based modeling languages.
4. Formal models are expensive to produce. Thus, reuse of at least parts of the language should be supported by the language design. Such features should also support the design of models for product families.
5. Accidents and major losses involving computers are usually the result of incompleteness or other flaws in the software requirements, not coding errors [8, 12]. We previously defined a set of formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [6, 8]. Engineers have made the criteria into checklists and used them on a variety of applications, such as radar systems, the Japanese module of the Space Station, and review criteria for FDA medical device inspectors. Two goals for SpecTRM-RL are to determine (1) how to enforce as many of the constraints as possible in the syntax of the language and (2) how to design the language to enhance the ability to manually check or build tools to automatically check the specifications for the criteria that cannot be enforced by the language design itself.

This paper describes what we have learned from our experimentation with the design of SpecTRM-RL about achieving the first four goals. Our results for the fifth goal will be described in a future paper. The design features of SpecTRM-RL that support each of these goals are described but a complete description of the language, including its syntax, is beyond the scope of this paper. We are currently producing a SpecTRM-RL language design manual and automated tools to assist in experimental use of the language.

2 Enhancing Usability and Reviewability

The primary goal for the design of a specification language should be to make the representation appropriate for the tasks to be performed by the users, i.e., to make the design *user-centered* (rather than designed primarily to make analysis easier or to be faithful to standard mathematical conventions). Software is a human product and specification languages are used to help humans perform the various problem-solving activities involved in software engineering. Our goal is to provide specifications that support human problem-solving and the tasks that humans must perform in software development and evolution as well as to allow automated analysis. We attempt to support human problem-solving by grounding specification design on psychological principles of how humans use specifications to solve problems as well as on basic system engineering principles.

We discuss two of these aspects here: minimizing semantic distance (problem 1 above) and building blackbox specifications (problem 2 above). Problems 3 and 4, as they reflect on the design of SpecTRM-RL, are discussed in later sections of this paper.

An important psychological principle for enhancing reviewability is the concept of semantic distance [5]. We define an informal concept of *semantic distance*, similar to Norman's use of the term, as the amount of effort required to translate from one model to another. We believe that in order to maximize the application expert's ability to find errors in a requirements specification, the semantic distance between their understanding of the required process control behavior (their mental model of the system) and the specification of that behavior must be minimized. This, in turn, implies that the requirements be written entirely in terms of the components and state variables of the controlled system. Specifically, "private" variables and procedures (functions) related only to the implementation of the requirements and not part of the application expert's view of the controlled system should not be used. That is, the specification should be black box.

A blackbox model of behavior permits statements and observations to be made only in terms of outputs and the inputs that stimulate or trigger those outputs. The model does not include any information about the internal design of the component itself, only its externally visible behavior. The overall system behavior is described by the combined behavior of the components, and the system design is modeled in terms of these component behavior models and the interactions and interfaces among the components.

When the description of the required controller behavior includes more than just its blackbox behavior (e.g., it includes software design information), then the semantic distance between the required process-control behavior and the specified controller behavior increases and the relationship between them becomes more difficult to validate (d_1 vs. d_4 in Figure 1). In fact, if adequately efficient code can be generated from the requirements specification directly, then an internal design specification may never be needed. "Adequately efficient" must be determined for each specific application's timing requirements. We are working on this code-generation problem [7].

In addition, the requirements review process involves validating the relationship between changes in the real-world process and the specified changes and response in the control function model. Therefore, reviewability will be enhanced if the requirements specification explicitly shows this relationship. We discuss this further in the next section.

Blackbox requirements specification languages not only enhance readability and reviewability, but they also simplify the transition from system requirements and system design to software requirements. The gap between system design and software requirements is frequently cited as a major problem in our interactions with industry. We believe some of the problem stems from the fact that software requirements often contain a lot of software design decisions, which makes the gap between the two specifications larger and more complex to negotiate.

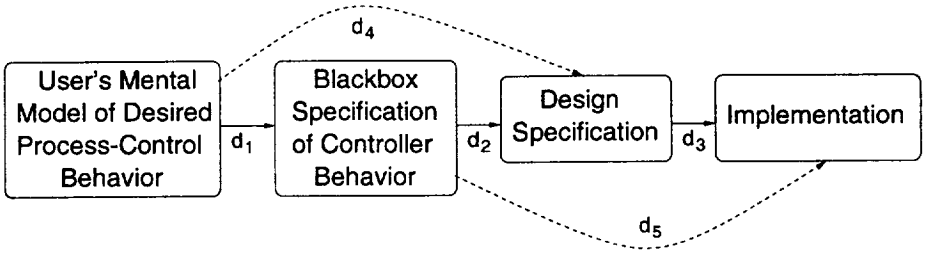


Fig. 1. Reviewability increases as the semantic distance between the user's mental model of the desired behavior and the specification (d_1 vs. d_4) decreases.

2.1 Minimizing Semantic Distance

Our language is designed primarily for process-control systems. Therefore we attempt to minimize the semantic distance d_1 by basing the specification language design on fundamental process control principles. In process control, the goal is to maintain a particular relationship or function F over time (t) between the input to the system $\mathcal{I}_s$ and the output from the system $\mathcal{O}_s$ in the face of disturbances $\mathcal{D}$ in the process (see Figure 2). This system function consists of the functional description and the set of constraints on the system [11]. At any moment, there is a unique set of relationships between inputs and outputs whereby each output value will be related to the past and present values of the inputs and time. These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic, or other laws as embodied within the nature and construction of the system. The system is constructed from components whose interaction implements F including, usually, a controller or controllers whose function is to ensure that F is correctly achieved.

A typical process-control system can be divided into four types of components: the process, sensors, actuators, and controller (see Figure 2).

Process: The behavior of the *process* is monitored through *controlled variables* ($\mathcal{V}_c$) and controlled by *manipulated variables* ($\mathcal{V}_m$). The process can be described by the process function F_P, a mapping from $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$.

Sensors: These devices are used to monitor the actual behavior of the process by measuring the controlled variables. For example, a thermometer may measure the temperature of a solvent in a chemical process or a barometric altimeter may measure altitude of an aircraft above sea level. The sensor function F_S maps $\mathcal{V}_c \times t \rightarrow \mathcal{I}$.

Actuators: These are devices designed to manipulate the behavior of the process, e.g., valves controlling the flow of a fluid or a pilot changing the direction and speed of an aircraft. The actuators physically execute commands issued by the controller in order to change the manipulated variables. The functionality of the actuators is described by the actuator function F_A mapping $\mathcal{O} \times t \rightarrow \mathcal{V}_m$.

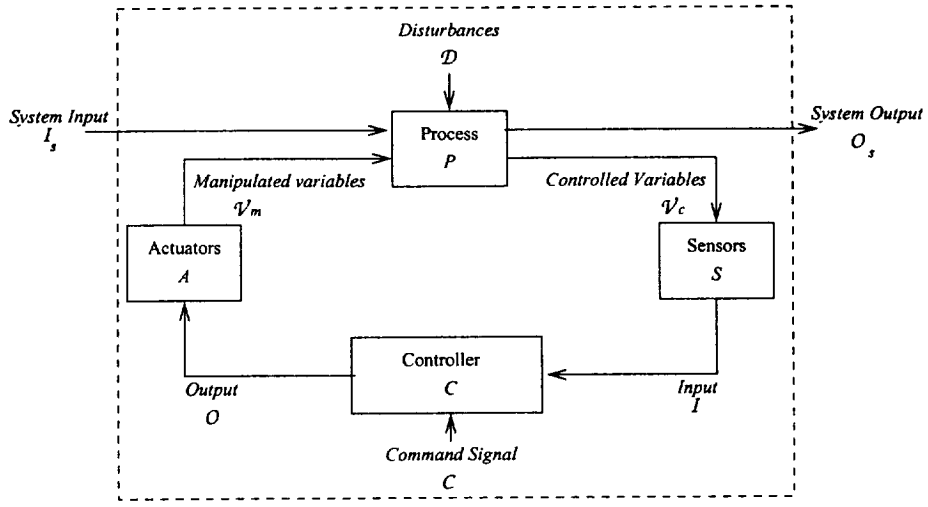


Fig. 2. Basic Process Control Loop

Controller: The *controller* is an analog or digital device used to implement the control function. The functional behavior of the controller is described by a control function (F_C) mapping $\mathcal{I} \times \mathcal{C} \times t \rightarrow \mathcal{O}$, where $\mathcal{C}$ denotes external command signals. The process may change state not only through internal conditions and through the manipulated variables, but also by disturbances ($\mathcal{D}$) that are not subject to adjustment and control by the controller. The general control problem is to adjust the manipulated variables so as to achieve the system goals despite disturbances. *Feedback* is provided via the controlled variables in order to monitor the behavior of the process. This feedback information (along with external command signals C) can be used as a foundation for future control decisions as well as an indicator of whether the changes in the process initiated by the controller have been achieved.

To reason about this type of process-controlled system, Parnas and Madey defined what they call the four-variable model [14]. This model is essentially an abstraction of part of the traditional feedback process control model presented here. The approach to modeling used in Parnas Tables [13] and SCR [4, 3] are based on this four variable model and, thus, built upon this same classic process control model.

The model presented in this section is an abstraction—responsibility for implementing the control function may actually be distributed among several components including analog devices, digital computers, and humans. Furthermore, the controller may have only partial control over the process—state changes in the process may occur due to internal conditions in the process or because of external disturbances or the actuators may not perform as expected. For example, the pilot in a TCAS system may not follow the resolution advisory (escape maneuver) issued by the TCAS controller.

The purpose of a control system requirements specification is to define the system goals and constraints, the function F_C (i.e., the required blackbox behavior of the controller), and the assumptions about the other components of the process-control loop that (1) the implementors need to know in order to implement the control function correctly and (2) the system engineers and analysts need to know in order to validate the model against the system goals and constraints.

A blackbox, behavioral specification of such a system function F_C uses only:

(1) the current process state inferred from measurements of the controlled variables,
(2) past process states that were measured and inferred,
(3) past corrective actions output from the controller, and
(4) prediction of future states of the controlled process

to generate the corrective actions (or current outputs) needed to maintain F.

All of this information can be embedded in a state-machine model of the controlled process, and we specify the blackbox behavior of the controller (i.e., the function F_C to be computed by the controller) using such a state machine model. In SpecTRM-RL models, the outputs of the controller are specified with respect to state changes in the model as information is received about the current state of the controlled process via the controlled variables $\mathcal{V}_c$. In the TCAS example, the control function is specified using a model of the state of all other aircraft within the host aircraft's airspace, the state of the on-board components of its own aircraft (e.g., altimeters, aircraft discretes[1], cockpit displays), and the state of ground-based radar stations in the vicinity. Information about this state is received from the sensors (e.g., antennas and transponders) and commands are sent to the actuators (e.g., the pilot and transponders).

The state machine model of the control function must be iteratively fine tuned during requirements development to mimic the current understanding of the real-world process and the required controller behavior. The state machine is essentially an abstraction of the behavior of the system function because it models all the relevant aspects of the components of the process control loop. Errors in the state machine model represent mismatches between this model and the desired behavior of the control loop, including the process.

3 Building Blackbox Specifications in SpecTRM-RL

Although RSML allows blackbox behavior specifications, the language itself does not encourage or enforce them. We found people tend to include design in the specification when using general state-machine modeling languages such as RSML or Statecharts. SpecTRM-RL, therefore, was not designed to be a general modeling language, but rather specifically designed to create blackbox

[1] Aircraft discretes are airframe-specific characteristics provided as input to TCAS from hardware switches.

requirements specifications to define an input/output process-control function, as is also true of SCR and Parnas Tables. General modeling features not needed for blackbox specifications are not included in SpecTRM-RL, and new abstractions (such as modes) are included that assist in blackbox modeling of control system components. Thus, SpecTRM-RL is not just another variant of Statecharts although there are some superficial similarities. Like SCR and Parnas Tables, SpecTRM-RL enforces the specification of an input/output process control function. Statecharts allows much more general models to be built.

In order to make our language formal enough to be analyzable (and yet readable and reviewable by non-mathematicians), we have defined a formal model (RSM or Requirements State Machine) that underlies a more readable specification language or languages. The RSM is a general behavioral model of the required control function with the components of the state machine mapped to the appropriate components of the control loop. This model has been published previously [6], and we do not refer to it further in this paper. We note only that the underlying model is a Mealy automaton, as is the model for SCR, Parnas Tables, Statecharts, and most other languages based on state-machines.

The higher-level specification language based on this underlying model must allow the modeler to specify the required process-control function F_C. Figure 3 shows a more detailed view of the components of the control loop, including distinguishing between human and automated controllers.

All control software (and any controller in general) uses an internal model of the general behavior and current state of the process that it is controlling. This internal model may range from a very simple model including only a few variables to a much more complex model with a large number of state variables and transitions. The model may be embedded in the control logic of an automated controller or in the mental model maintained by a human controller. It is used to determine what control actions are needed. The model is updated and kept consistent with the actual system state through various forms of feedback.

The design of SpecTRM-RL is influenced by our desire to perform safety analysis on the models. When the controller's model of the system diverges from the actual system state, erroneous control commands (based on the incorrect model) can lead to an accident—for example, the software does not know that the plane is on the ground and raises the landing gear or it does not identify an object as friendly and shoots a missile at it. The situation becomes more complicated when there are multiple controllers (both human and automated) because their internal system models must also be kept consistent. In addition, human controllers interacting with automated controllers must also have a model of the automated controllers' behavior in order to monitor or supervise the automation as well as the controlled system itself.

One reason the models may diverge is that information about the process state has to be inferred from measurements. For example, in TCAS, relative range positions of other aircraft are computed based on round-trip message propagation time. Theoretically, the function F_C can be defined using only the true values of the controlled variables or component states (e.g., true aircraft po-

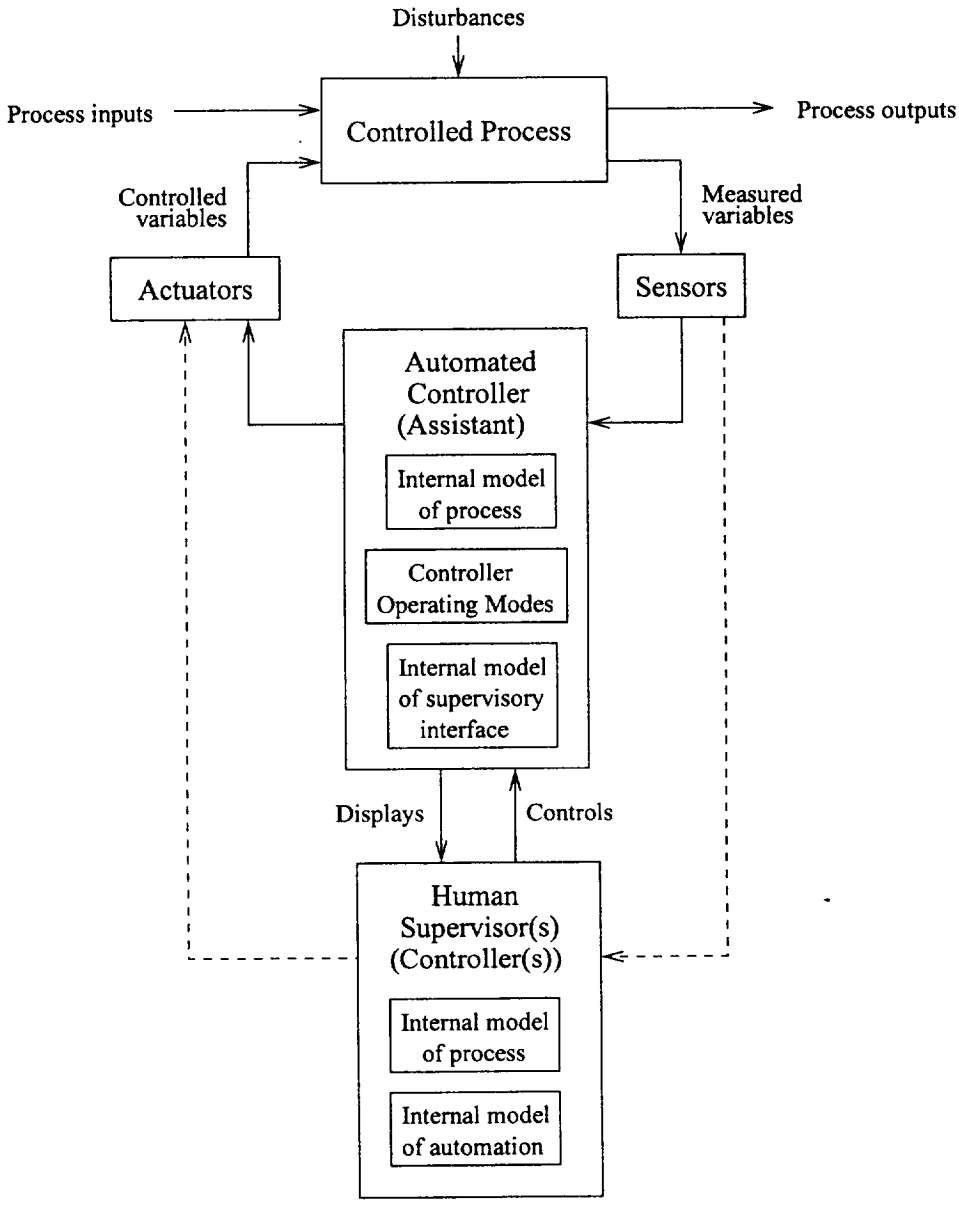


Fig. 3. A basic control loop. A blackbox requirements specification captures the controller's internal model of the process. Accidents occur when the internal model does not accurately reflect the state of the controlled process.

sitions). However, at any time, the controller has only measured values of the component states (which may be subject to time lags or measurement inaccuracies), and the controller must use these measured values to infer the true conditions in the process and possibly to output corrective actions ($\mathcal{O}$) to maintain F. In the TCAS example, sensors include on-board devices such as altimeters that provide measured altitude (not necessarily true altitude) and antennas for communicating with other aircraft. The primary TCAS actuator is the pilot, who may or may not respond to system advisories. Pilot response delays are important time lags that must be considered in designing the control function. Time lags in the controlled process (the aircraft trajectory) may be caused by aircraft performance limitations.

The automated controller also has a model of its interface to the human controllers or its supervisor(s). This interface, which contains the controls, displays, alarm annunciators, etc. is important because it is the means by which the two controllers' models are synchronized. Each of these components is included explicitly in our models and modeling language. We represent the controlled process and supervisory interface models using state machines and define required behavior in terms of transitions in this machine. The controller outputs (commands to the actuators) are specified with respect to state changes in the model as information is received about the current state of the controlled process via controlled variables read by sensors.

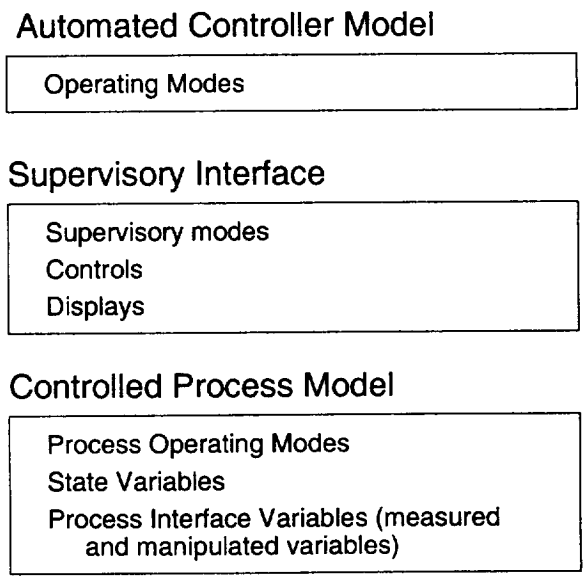


Fig. 4. A SpecTRM-RL model has three parts.

Thus a SpecTRM-RL specification of control software is composed of three interrelated models (Figure 4): (1) a specification of the operating modes of the controller, (2) a specification of the controllers's view of its supervisory interface (the component or components, including human operators, that are controlling it), and (3) a model of the controlled process.

3.1 The Structure of a SpecTRM-RL Specification

Engineers often use modes in describing required system functionality. Mode confusion also is frequently implicated in the analysis of operator errors that lead to accidents. We have included in SpecTRM-RL the ability to describe behavior in terms of modes both to reduce semantic distance (and enhance reviewability) and to allow for analysis of various types of mode-related errors [10].

A mode can be defined as a mutually exclusive set of system behaviors. For example, the following table shows the possible transitions between states in a simple state machine given two system modes: startup mode and normal operation mode.

	s_1	s_2	s_3	s_4	s_5
Startup mode	s_3	s_2	s_4	s_5	s_1
Normal mode	s_3	s_4	s_1	s_5	s_1

Table 1. A simple state machine with two modes defined using a standard state transition table. The states in the machine (listed at the top of the table) are s_1 through s_5 while the conditions under which the transition is made are listed on the left (e.g., startup mode and normal mode). Note that transitions may depend on more conditions than simply the current processing mode.

The startup and normal processing modes in this machine determine how the machine will behave over the entire set of state transitions. For example, if the conditions occur that trigger a transition from state s_3, the machine will transfer to state s_4 if it is in startup mode or to state s_1 if it is in normal processing mode. Note that modes are simply states that play a particular role in the state machine behavior (i.e., control a sequence or set of state transitions). That is, they are a convenient abstraction for describing and understanding complex system behavior, but they do not add any power to the state machine description. In general, state transitions may be triggered by events, conditions, or simply the passage of time. The current operating mode determines how these triggers will be interpreted and what transitions will be taken. Note that there is no real difference between a state and a mode by this definition. Any conditions or states could be labelled a "mode" (which indeed is done in some specification languages), although this is not very helpful and is not the way engineers use the term "mode".

Modern aircraft and other complex systems often have a large number of operating modes and possible combinations of operating modes. In the modeling and analysis of control systems, we find it useful to distinguish between three different types of modes:

1. *Supervisory modes* determine who or what is controlling the component at any time. Control loops may be organized hierarchically, with multiple controllers or components, each being controlled by the layer above and controlling the layer below. In addition, each component may have multiple controllers (supervisors). For example, a flight control computer in an aircraft may get inputs from the flight management computer or directly from the pilot. Required behavior may be different depending on what supervisory mode is currently in effect. Mode-awareness errors related to confusion in coordination between the multiple supervisors of a control component can be defined in terms of these supervisory modes.
2. *Component operating modes* control the behavior of the control component itself. They may be used to control the interpretation of the component's interfaces or to describe the component's required process-control behavior.
3. *Controlled-system* (or *plant* in control theory terminology) *operating modes* specify sets of related behaviors of the controlled system and are used to indicate its operational state. For example, an aircraft may be in takeoff, climb, level-flight, descent, or landing mode.

The use of modes does not violate the blackbox nature of SpecTRM-RL; they represent externally visible behavior (required functionality) of the component and not the internal design of the software to achieve that functionality. For example, *capture mode* (which can be *armed* or *not armed*) in the flight management system example shown in Figure 5 indicates whether the aircraft will automatically level off when a pilot-specified altitude constraint is reached. The pilot is responsible for setting the altitude constraint and (usually) for directly or indirectly selecting capture mode.

As stated earlier, a SpecTRM-RL specification has three interrelated models. The top box of Figure 5 shows the graphical part of an example specification of a flight management system. The system has seven modes of operation, all of which have only one value at any one time. The boxes shown under each mode label represents the discrete values for that mode, e.g., pitch can be in *altitude hold, vertical speed, indicated air speed*, or *altitude capture* mode. The line at the left of the choices simply groups the choices under the variable and indicates that only one may be active at any time and does not represent state transitions (as it did in RSML).

A second part of a SpecTRM-RL model is a specification of the component's view of its supervisory interface. The supervisory interface consists of a model of the operator controls and displays or other means of communication by which the component relays information to the supervisor. Note that the interface models are simply the view that the component has of the interfaces—the real state of the interface may be inconsistent with the assumed state due to various types of design flaws or failures. For example, a flight control computer in an aircraft may get inputs from the flight management computer or directly from the pilot. Required behavior may be different depending on what supervisory mode is currently in effect. By separating the assumed interface from the real interface, we are able to model and analyze the effects of various types of errors and

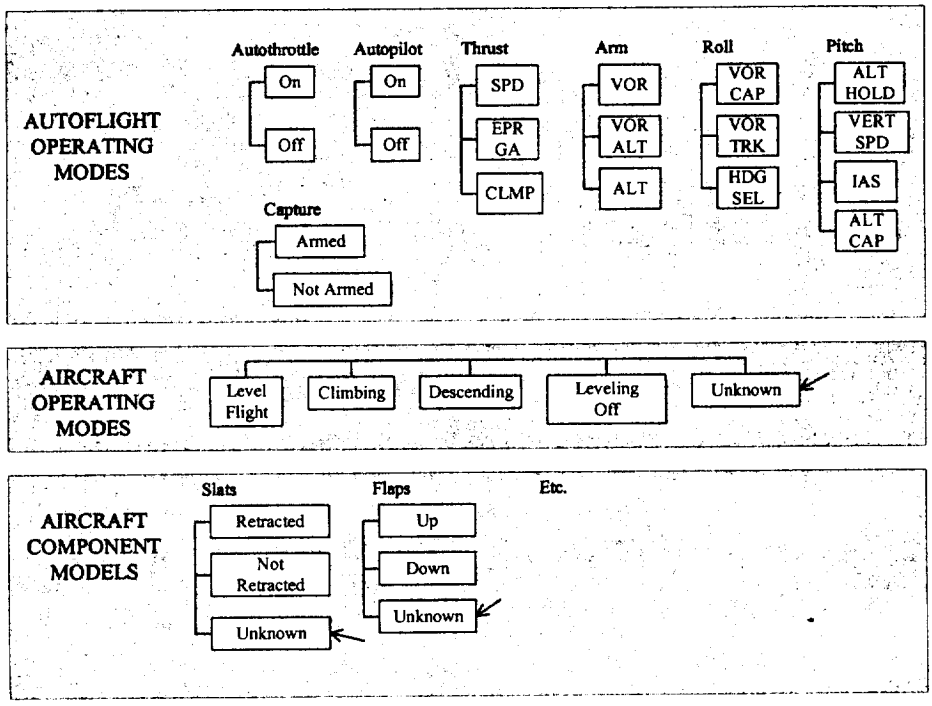


Fig. 5. Example of operating modes for a flight management system

failures (e.g., communication errors or display hardware failures). In addition, separating the physical design of the interface from the logical design (required content) will facilitate changes and allow parallel development of the software and the interface design.

The third part of a SpecTRM-RL model is the component's model of the controlled system (plant). The description of a simple component may include only a few relevant state variables. If the controlled process or component is complex, the model of the controlled process may itself be represented in terms of its operational modes and the states of its subcomponents. In a hierarchical control system, the controlled process may itself be a controller of another process. For example, the flight management system may be controlled by a pilot and may itself issue commands to a flight control computer, which issues commands to an engine controller. If, during the design process, components that already exist are used, then those plug-in component models could be inserted into the SpecTRM-RL process model.

If the SpecTRM-RL model is of a non-control component (e.g., a radar data processor), there might not be a supervisory interface. There will still be operating modes, however, and a model of the required input-output function to be computed by the component.

The language itself consists of a graphical model of the state machine, output message specifications, state variable definitions, operator interface variable definitions, state transition specifications, macros, and functions.

Graphical State Machine. The SpecTRM-RL notation is driven by the use of the language to define a function from outputs to inputs. SpecTRM-RL has a greatly simplified graphical representation (compared to RSML or Statecharts), which is made possible because we eliminated the types of state machine complexity necessarily for specifying component design but not necessary to specify the input/output function computed in a pure blackbox requirements specification. The architecture of the state transitions becomes so simple that we found no need to represent it in the graphical state machine—the transitions simply represent the changes between state variable values.

State values in square boxes represent *inferred values*. Inferred values are not input directly but represent the aspects of the process state model that must be inferred from measurements of monitored process variables. Inferred process states are used to control the computation of the control function. They are necessarily discrete in value[2], and thus can be represented as a finite state variable. In practice, such state variables almost always have only a few relevant values (e.g., altitude below 500 feet, altitude between 500 feet and 10,000 feet, altitude above 10,000 feet). State values denoted as circles or ovals represent direct input and output values (controlled or monitored variables).

[2] If they are not discrete, then they are not used in the control of the function computation but in the computation itself and can simply be represented in the specification by arithmetic expressions involving input variables.

The supervisory interface model shows the supervisory mode, which describes how this computer is being supervised, e.g., by a human, computer, or both (Figure 6). It also shows the state of the controls and the displays (including oral annunciations, etc.).

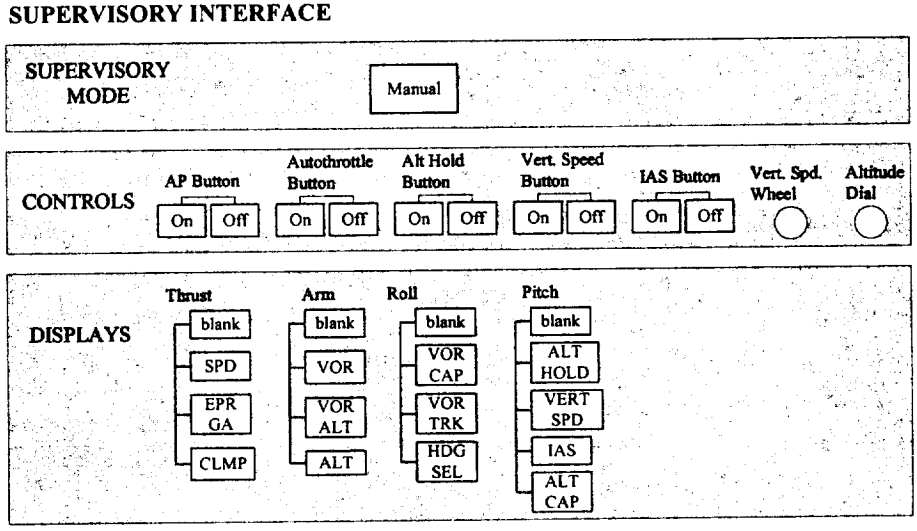


Fig. 6. Example of SpecTRM-RL model of the supervisory modes

Output Message Specification. Everything starts from outputs in SpecTRM-RL. By starting from the output specification, the specification reader can determine what inputs trigger that output and the relationship between the inputs and outputs. RSML did not explicitly show this relationship (although it could be determined, with some effort, by examining the specification). A simplified example is shown in Figure 7. More information is actually required by our completeness criteria than is shown in the example, for instance, specification of timing assumptions related to the message.

The conditions under which an output is triggered (sent) is simply a predicate logic statement over the various states, variables, and modes in the specification. During the TCAS project, we discovered that the triggering conditions required to accurately capture the requirements were often extremely complex. The propositional logic notation traditionally used to define these conditions did not scale well to complex expressions and quickly became unreadable (and error-prone). To overcome this problem, we decided to use a tabular representation of disjunctive normal form (DNF) that we call AND/OR tables. We have maintained this successful notation in SpecTRM-RL. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunc-

tion of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated predicates. A dot denotes "don't care."

Output Message

Resolution Advisory

TRIGGERING CONDITION

Composite-RA$_{s\text{-}266}$ **in state** RA	T
Traffic-Display-Status$_{s\text{-}148}$ **in state** Waiting-To-Send	T

MESSAGE CONTENTS

FIELD	VALUE
Bits 11-17	Own-Goal-Altitude-Rate$_{v\text{-}219}$
Bits 18-20	Combined-Control$_{v\text{-}227}$
Bits 21-23	Vertical-Control$_{v\text{-}231}$
Bits 24-26	Climb-RA$_{v\text{-}233}$
Bits 27-29	Descent-RA$_{v\text{-}235}$

Fig. 7. Example of SpecTRM-RL output message specification

The subscripts in the specification represent whether the value is a variable (v) or a state (s). The other alternatives, macros (m) and functions (f) are described later in this paper. The number attached to the subscript is the page on which the variable, state, macro, or function is defined.

State Variable Definition. State variable values come from inputs or they may be computed from such input values or inferred from other state variable values. Figure 8 shows a partial example of a state variable description. Again, our desire to enforce completeness requires that state variable definitions include such information as arrival rates, exceptional condition handling, data age requirements, feedback information, etc. not shown here.

SpecTRM-RL requires all state variables that describe the process state to include an *unknown* value. This value is the default value upon startup or upon specific mode transitions (for example, after temporary shutdown of the computer). This feature is used to ensure consistency between the computer model of the process state and the real process state by forcing resynchronization of the model with the outside world after an interruption of processing. Many accidents have been caused by the assumption that the process state does not change

State

Supervisory-Interface

↳Pilot-Displays

↳ Resolution-Advisory

↳ Vertical-Control

DEFINITION

= Blank

INITIALLY

= Other

Some RA-Strength$_{s\text{-}277}$ **in state** Increase-2500fpm	F	F	F
Some Reversal$_{s\text{-}282}$ **in state** Reversed	F	F	F
Composite-RA$_{s\text{-}266}$ **in state** Climb	F	•	•
Composite-RA$_{s\text{-}266}$ **in state** Descend	F	•	•
Corrective-Climb$_{s\text{-}263}$ **in state** Yes	•	F	•
Corrective-Descend$_{s\text{-}264}$ **in state** Yes	•	•	F
Crossing-Geometry$_{m\text{-}388}$	F	F	F

= Increase

Some RA-Strength$_{s\text{-}277}$ **in state** Increase-2500fpm	T
Climb-Inhibit$_{s\text{-}243}$ **in mode** Inhibited	T
Descend-Inhibit$_{s\text{-}245}$ **in mode** Inhibited	T

= Crossing

Some Reversal$_{s\text{-}282}$ **in state** Reversed	F	F	F	F
Composite-RA$_{s\text{-}266}$ **in state** Climb	T	T	•	•
Composite-RA$_{s\text{-}266}$ **in state** Descend	•	•	T	T
Some RA-Strength$_{s\text{-}277}$ **in state** Increase-2500fpm	F	F	F	F
Corrective-Climb$_{s\text{-}263}$ **in state** Yes	F	•	F	•
Corrective-Descend$_{s\text{-}264}$ **in state** Yes	•	F	•	F
Crossing-Geometry$_{m\text{-}388}$	F	F	F	F

= Maintain

Composite-RA$_{s\text{-}266}$ **in state** Climb	T	•
Some RA-Strength$_{s\text{-}277}$ **in state** Increase-2500fpm	F	F
Composite-RA$_{s\text{-}266}$ **in state** Descend	•	T
Corrective-Climb$_{s\text{-}263}$ **in state** Yes	F	F
Corrective-Descend$_{s\text{-}264}$ **in state** Yes	F	F

= Reversal

Some Reversal$_{s\text{-}282}$ **in state** Reversed	T	T	T
Composite-RA$_{s\text{-}266}$ **in state** Climb	F	•	•
Composite-RA$_{s\text{-}266}$ **in state** Descend	F	•	•
Corrective-Climb$_{s\text{-}263}$ **in state** Yes	T	•	T
Corrective-Descend$_{s\text{-}264}$ **in state** Yes	•	•	T

Fig. 8. Example of SpecTRM-RL state variable specification

while the computer is not processing inputs or by incorrect assumptions about the initial value of state variables.

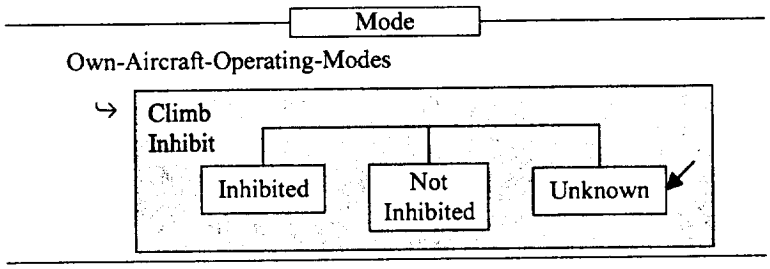
Unknown is used for state variables in the supervisory interface model only if the state of the display can change independently of the software. Otherwise, such variables must specify an initial value (e.g., blank, zero, etc.) that should be sent when the computer is restarted.

In the example shown, *vertical control* is a state variable in the supervisory interface model and is one of the pieces of information displayed to the pilot as part of an RA (Resolution Advisory, which is the escape maneuver the pilot is to implement to avoid the intruder aircraft). Vertical control can have the values *Unknown, Other, Increase, Crossing, Maintain,* or *Reversal.* AND/OR tables are used to specify which of these values is displayed to the pilot (given the current state of the aircraft model and the intruder aircraft being avoided). For example, *Maintain* is displayed if the Composite-RA state variable is in state "Climb", there is no RA-Strength in state "Increase-2500fpm", and Corrective-Climb and Corrective-Descend are both not in state "yes" *or* Maintain is displayed if there is no RA-Strength in state Increase-2500fpm, Composite RA is in state Descend, and again both Corrective-Climb and Corrective-Descend are not in state "yes." Timing constraints may also be specified as conditions in the tables (i.e., conditions on the state transitions) but are not required in this example.

State Transition Specification. As with all state-machine models, transitions in the three parts of a SpecTRM-RL model are governed by external events and the current state of the modeled system. In SpecTRM-RL, the conditions under which transitions are taken are specified separately from the graphical depiction of the state machine. We have found that the behavior of real systems is too complex to write on a line between two boxes. Instead, we again use AND/OR tables. Figure 9 shows an example specification for a transition.

Macros and Functions. Macros are simply named pieces of AND/OR tables that can be referenced from within another table. For example, the macro in Figure 10 is used in the definition of the variable Vertical-Control in Figure 8. The macros, for the most part, correspond to typical abstractions used by application experts in describing the requirements and therefore add to the understandability of the specification. In addition, the abstraction is necessary to handle the complexity of guarding conditions in larger systems and we found this a convenient abstraction to allow hierarchical review and understanding of the specification. Also, rather than including complex mathematical functions directly in the transition tables, functions are specified separately and referenced in the tables. For instance, Own-Tracked-Alt in Figure 9 is a function reference.

The macros and function, as well as the concept of parallel state machines, not only help structure a model for readability; they also help us organize models to enable *specification reuse.* Conditions commonly used in the application domain can be captured in macros and common functions, such as tracking, can be captured in reusable functions. In addition, the parallel state machines allow



DEFINITION

INITIALLY ⟶ Unknown

true

Unknown, Not-Inhibited ⟶Inhibited

Composite-RA$_{s\text{-}266}$ **in state** No-RA	T	T
Altitude-Climb-Inhibit$_{v\text{-}259}$ = True	T	•
Own-Tracked-Alt$_{f\text{-}487}$ > Aircraft-Altitude-Limit$_{v\text{-}259}$	T	•
Config-Climb-Inhibit$_{v\text{-}259}$ = True	•	T

Unknown, Inhibited ⟶Not-Inhibited

Composite-RA$_{s\text{-}266}$ **in state** No-RA	T	T
Altitude-Climb-Inhibit$_{v\text{-}259}$ = True	F	•
Own-Tracked-Alt$_{f\text{-}487}$ > Aircraft-Altitude-Limit$_{v\text{-}259}$	•	F
Config-Climb-Inhibit$_{v\text{-}259}$ = True	F	F

Fig. 9. Example of SpecTRM-RL mode or state transition specification

the internal model of each system component (discussed in Section 3) and the different system modes to be captured as separate and parallel state machines. This helps to accommodate reuse of internal models and operational modes, and helps us plan for product families (research goal 4 in the introduction). Naturally, to accomplish reuse, care has to be taken when creating the original model to determine what parts are likely to change and to modularize these parts so that substitutions can be easily made. This structuring, however, is beyond the scope of the current paper.

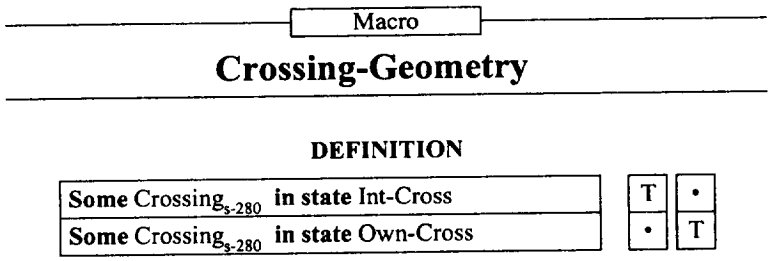


Fig. 10. Example of SpecTRM-RL macro specification

4 Eliminating Internal Broadcast Events

A third goal for SpecTRM-RL was to eliminate error-prone constructs. During the independent verification and validation (IV&V) of TCAS II, problems with internal broadcast events (used to synchronize parallel state machines in Statecharts and RSML) accounted for a clear majority of the errors related to the syntax and semantics of RSML. Common and difficult to resolve problems involved proper synchronization of mutually interdependent state machines. In addition, getting the state machines to correctly model system startup behavior proved to be surprisingly difficult. Internally generated events seem to cause "accidental complexity" in the specification that is not necessarily present in the problem being specified.

These problems were not just the most common language-related problems in the initial specification, they were also the problems that lingered unresolved (or incompletely resolved) through several cycles of corrections and repeated IV&V. Note that the problems related to synchronization were not directly caused by misunderstandings of the RSML event/action semantics; the event/action mechanism is quite simple and purposely selected to be intuitive [9]. Instead, the problems were caused by the complexity of the model and the inherent difficulty of comprehending the causal relationships between parallel state machines. Thus, this difficulty is not unique to RSML, it is fundamentally difficult to understand parallelism and synchronization. Other state-based languages such as Statecharts

[1] and the UML behavioral (state machine) models that use event/action semantics are likely to encounter the same problem when used to model complex systems. When we eliminated internal events, we were surprised at how much easier it was to rewrite our old specifications (such as TCAS II) and to create new ones.

The trigger events and actions on the transitions in Statecharts and RSML are used for two purposes. First, they are used to sequence and synchronize state machines so the next state relation is computed correctly. For instance, to determine if an intruding aircraft is a collision threat in TCAS II, we must first determine how close the intruder is and how close the intruder is allowed to come before it is considered a threat. Thus, the state variables determining the intruder status and the sensitivity level of TCAS II must be evaluated before we determine advisory-status. This is a straightforward (but as mentioned above, error prone) use of events and actions.

Second, events and actions may be employed to use, in essence, the state machines as a programming language. The events can be used to create loops and counters, and events can be implicitly assigned semantic meaning and used for purposes other than synchronization. In our experience we have found this freedom of using the events a trap that invites the introduction of design details in the specification. During the development of the TCAS II model we had to repeatedly remind ourselves to use events prudently; we have found that even experienced users of such modeling languages inevitably fall into the trap of using events and actions to introduce too much design in the models.

To solve this problem in SpecTRM-RL, we simply decided not to use internal events and instead to rely on the data dependencies in the model to determine the order in which transitions are taken, i.e., the ordering, if critical, is explicitly included in the model as opposed to being built into the semantics of the modeling language. In this way, the reviewer need not rely on knowledge about the semantics of the modeling language to determine if the model correctly matches the intended functional behavior—that behavior (which state transitions follow which) is explicitly specified in the constructed model. A similar argument holds for the modeler. We found that different reviewers of our TCAS specification were assigning differing semantics to the state transition ordering. In the example above, the transitions in the state machine advisory status refer to the states of intruder status and sensitivity level; thus, intruder status and sensitivity level will be evaluated before advisory status. This sequencing will assure a correct evaluation of the next state relation based on the data dependencies of the transitions and variables. The next state function is recomputed every time the environment changes an input variable. Naturally, a SpecTRM-RL specification cannot include any cycles in the data dependencies. Cycles in a specification can be easily detected by our tools.

In Statecharts and RSML, a transition is not taken until an explicit event is generated. When the transition is taken, additional events may be generated as actions. In this way, the events propagate through the state machine triggering transitions. In our formalization of the semantics of RSML [2] we view each

transition as a simple function mapping one state to the next. The events and actions on the transitions are used to determine in which order we shall use these functions to compute the next state. We define the new semantics of SpecTRM-RL in essentially the same way as for RSML. The only difference is how we determine in which order to apply the functions representing transitions. We now rely entirely on the data dependencies between the transitions to determine a partial order that is used during the computation of the next state relation. The semantics of SpecTRM-RL have been defined formally, but this definition is not included for space reasons.

5 Conclusions

In this paper, we described some lessons learned during experimentation with a formal specification language (RSML) and how we have used what we learned to drive further research. We showed how a formal modeling language can be designed to assist system understanding and the requirements modeling effort. We achieve this by grounding the design of the language in the domain for which it is intended (process control) and how people actually think about and conceptualize complex systems.

We have applied these principles to the design of a new experimental language called SpecTRM-RL. As mentioned above, SpecTRM-RL evolved from our previous experiences with using RSML to specify large and complex systems. In particular, we addressed the problems associated with inclusion of excessive design in the blackbox specification and internal broadcast events. Our experience thus far indicates that the new language design principles introduced in SpecTRM-RL greatly enhance the usability of a formal notation.

References

1. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.
2. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, pages 363–377, June 1996.
3. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions of Software Engineering and Methodology*, 5(3):231–261, July 1996.
4. K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.
5. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1:311–338, 1985.
6. Matthew S. Jaffe, Nancy G. Leveson, Mats P.E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

7. D.J. Keenan and M.P.E. Heimdahl. Code generation from hierarchical state machines. In *Proceedings of the International Symposium on Requirements Engineering*, 1997.
8. N.G. Leveson. *Safeware: System Safety and Computers.* Addison Wesley, 1995.
9. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, pages 684–706, September 1994.
10. N.G. Leveson, J.D. Reese, S. Koga, L.D. Pinnel, and S.D. Sandys. Analyzing requirements specifications for mode confusion errors. In *Proceedings of the Workshop on Human Error and System Development*, 1997.
11. E.I. Lowe. *Computer Control in Process Industries.* Peregrinus, 1971.
12. Robyn R. Lutz. Targeting safety related errors during software requirements analysis. *Journal of Systems Software*, 34(3):223–230, September 1996.
13. David L. Parnas. Tabular representations of relations. Technical Report CLR report No. 260, McMaster University, Hamilton, Ontario, October 1992.
14. David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.

To appear in IEEE Transactions on Software Engineering, Jan. 2000

Intent Specifications: An Approach to Building Human-Centered Specifications *

Nancy G. Leveson
Dept. of Computer Science and Engineering
University of Washington

Abstract. This paper examines and proposes an approach to writing software specifications, based on research in systems theory, cognitive psychology, and human-machine interaction. The goal is to provide specifications that support human problem solving and the tasks that humans must perform in software development and evolution. A type of specification, called *intent specifications*, is constructed upon this underlying foundation.

1 The Problem

Software is a human product and specification languages are used to help humans perform the various problem-solving activities involved in requirements analysis, software design, review for correctness (verification and validation), debugging, maintenance and evolution, and reengineering. This paper describes an approach, called intent specifications, to designing system and software specifications that potentially enhances human processing and use by grounding specification design on psychological principles of how humans use specifications to solve problems as well as on basic system engineering principles. Using such an approach allows us to design specification languages with some confidence that they will be usable and effective.

A second goal of intent specifications is to integrate formal and informal aspects of software development and enhance their interaction. While mathematical techniques are useful in some parts of the development process and are crucial in developing software for critical systems, informal techniques will always be a large part (if not most) of any complex software development effort: Our models have limits in that the actual system has properties beyond the model, and mathematical methods cannot handle all aspects of system development. To be used widely in industry, our approach to specification must be driven by the need (1) to systematically and realistically balance and integrate mathematical and nonmathematical aspects of software development and (2) to make the formal parts of the specification easily readable, understandable, and usable by everyone involved in the development and maintenance process.

Specifications should also enhance our ability to engineer for quality and to build evolvable and changable systems. Essential system-level properties (such as safety and security) must be built into the design from the beginning; they cannot be added on or simply measured afterward. Up-front planning and changes to the development process are needed to achieve particular objectives. These changes include using notations and techniques for reasoning about particular properties, constructing the system and the software in it to achieve them, and validating (at each step, starting from the very beginning of system development) that the evolving system has the desired qualities. Our specifications must reflect and support this process. In addition, systems and software are continually changing and evolving; they must be designed to be changeable and the specifications must support evolution without compromising the confidence in the properties that were initially verified.

Many of the ideas in this paper are derived from attempts by cognitive psychologists, engineers, and human factors experts to design and specify human–machine interfaces. The human-machine interface provides a representation of the state of the system that the operator can use to solve problems and perform control, monitoring, and diagnosis tasks. Just as the control panel in a plant is the interface between the operator and the plant, system and software requirements and design specifications are the interface between the system designers and builders or builders and maintainers. The specifications help the designer, builder, tester, debugger, or maintainer understand the system well enough to create a physical form or to find problems in or change the physical form.

The paper is divided into two parts. The first part describes some basic ideas in systems theory and cog-

*This work was partially supported by NASA Grants NAG-1-1495 and NAG-1-2020 and by NSF Grant CCR-9396181.

nitive engineering[1]. The second part describes a type of specification method called *intent specifications* built upon these basic ideas that is designed to satisfy the goals listed above, i.e., to enhance human processing and problem solving, to integrate formal and informal aspects of software development, and to enhance our ability to engineer for quality and to build evolvable and changeable systems.

2 Specifications and Human Problem Solving

To be useful to and usable by humans to solve problems, specification language and system design should be based on an understanding of the problem or task that the user is solving. The systems we design and the specifications we use impose demands on humans. We need to understand those demands and how humans use specifications to solve problems if we are to design specifications that reflect reasonable demands and that assist humans in carrying out their tasks.

Not only does the language in which we specify problems have an effect on our problem-solving ability, it also affects the errors we make while solving those problems. Our specification language design needs to reflect what is known about human limitations and capabilities.

A problem-solving activity involves achieving a goal by selecting and using strategies to move from the current state to the goal state. Success depends on selecting an effective strategy or set of strategies and obtaining the information necessary to carry out that strategy successfully. Specifications used in problem-solving tasks are constructed to provide assistance in this process. Cognitive psychology has firmly established that the representation of the problem provided to problem solvers can affect their performance (see Norman [Nor93] for a survey of this research). In fact, Woods claims that there are no neutral representations [Woo95]: The representations available to the problem solver either degrade or support performance. To provide assistance for problem solving, then, requires that we develop a theoretical basis for deciding which representations support effective problem-solving strategies. For example, problem-solving performance can be improved by providing representations that reduce the problem solver's memory load [KHS85] and that display the critical attributes needed to solve the problem in a perceptually salient way [KS90].

A problem-solving strategy is an abstraction describing one consistent reasoning approach characterized by a particular mental representation and interpretation of observations [RP95]. Examples of strategies are hypothesis and test, pattern recognition, decision tree search, reasoning by analogy, and topological search.

Some computer science researchers have proposed theories about the mental models and strategies used in program understanding tasks (examples of such models are [Bro83, Let86, Pen87, SM79, SE84]). Although this approach seems useful, it may turn out to be more difficult than appears on the surface. Each of the users of a specification may (and probably will) have different mental models of the system, depending on such factors as prior experience, the task for which the model is being used, and their role in the system [AT90, Dun87, Luc87, Rea90]. The same person may have multiple mental models of a system, and even having two contradictory models of the same system does not seem to constitute a problem for people [Luc87]

Strategies also seem to be highly variable. A study that used protocol analysis to determine the troubleshooting strategies of professional technicians working on electronic equipment found that no two sequences of actions were identical, even though the technicians were performing the same task every time (i.e., finding a faulty electronic component) [Ras86]. Not only do search strategies vary among individuals for the same problem, but a person may vary his or her strategy dynamically during a problem-solving activity: Effective problem solvers change strategies frequently to circumvent local difficulties encountered along the solution path and to respond to new information that changes the objectives and subgoals or the mental workload needed to achieve a particular subgoal.

It appears, therefore, that to allow for multiple users and for effective problem solving (including shifting among strategies), specifications should support all possible strategies that may be needed for a task to allow for multiple users of the representation, for shedding mental workload by shifting strategies during problem solving, and for different cognitive and problem-solving styles. We need to design specifications such that users can easily find or infer the information they need regardless of their mental model or preferred problem-solving strategies. That is, the specification design should be related to the general tasks users need to perform with the information but not be limited to specific predefined ways of carrying out those tasks.

One reason why many software engineering tools and environments are not readily accepted or easily used is

[1] *Cognitive engineering* is a term that has come to denote the combination of ideas from systems engineering, cognitive psychology, and humans factors to cope with the challenges of building high-tech systems composed of humans and machines. These challenges have necessitated augmenting traditional human factors approaches to consider the capabilities and limitations of the human element in complex systems.

that they imply a particular mental model and force potential users to work through problems using only one or a very limited number of strategies, usually the strategy or strategies preferred by the designer of the tool. The goal of specification language design should be to make it easy for users to extract and focus on the important information for the specific task at hand without assuming particular mental models or limiting the problem-solving strategies employed by the users of the document. The rest of this paper describes an approach to achieve this goal.

3 Components of a Specification Methodology to Support Problem-Solving

Underlying any methodology is an assumed *process*. In our case, the process must support the basic system and software engineering tasks. A choice of an underlying system engineering process is the first component of a specification methodology. In addition, cognitive psychologists suggest that three aspects of interface design must be addressed if the interface is to serve as an effective medium: (1) *content* (what semantic information should be contained in the representation given the goals and tasks of the users, (2) *structure* (how to design the representation so that the user can extract the needed information), and (3) *form* (the notation or format of the interface) [VR90]. The next sections examine each of these four aspects of specification design in turn.

3.1 Process

Any system specification method should support the systems engineering process. This process provides a logical structure for problem solving (see Figure 1). First a need or problem is specified in terms of objectives that the system must satisfy and criteria that can be used to rank alternative designs. Then a process of system synthesis takes place that results in a set of alternative designs. Each of these alternatives is analyzed and evaluated in terms of the stated objectives and design criteria, and one alternative is selected to be implemented. In practice, the process is highly iterative: The results from later stages are fed back to early stages to modify objectives, criteria, design alternatives, and so on.

Design alternatives are generated through a process of system architecture development and analysis. The system engineers break down the system into a set of subsystems, together with the functions and constraints imposed upon the individual subsystem designs, the major system interfaces, and the subsystem interface topology. These aspects are analyzed with respect to desired system performance characteristics and constraints, and the process is iterated until an acceptable system design results. The preliminary design at the end of this process must be described in sufficient detail that subsystem implementation can proceed independently.

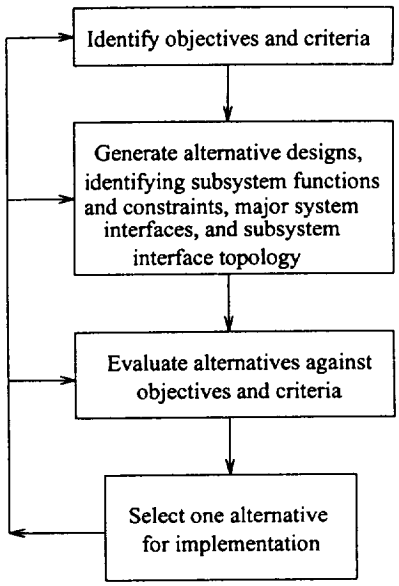


Figure 1: The basic systems engineering process.

The software requirements and design process are simply subsets of the larger system engineering process. System engineering views each system as an integrated whole even though it is composed of diverse, specialized components, which may be physical, logical (software), or human. The objective is to design subsystems that when integrated into the whole provide the most effective system possible to achieve the overall objectives. The most challenging problems in building complex systems today arise in the interfaces between components. One example is the new highly automated aircraft where most incidents and accidents have been blamed on human error, but more properly reflect difficulties in the collateral design of the aircraft, the avionics systems, the cockpit displays and controls, and the demands placed on the pilots.

What types of specifications are needed to support humans in this system engineering process and to specify the results? Design decisions at each stage must be mapped into the goals and constraints they are derived to satisfy, with earlier decisions mapped (traced) to later stages of the process, resulting in a seamless (gapless) record of the progression from high-level system requirements down to component requirements and

designs. The specifications must also support the various types of formal and informal analysis used to decide between alternative designs and to verify the results of the design process. Finally, they must assist in the coordinated design of the components and the interfaces between them.

3.2 Content

The second component of a specification methodology is the content of the specifications. Determining appropriate *content* requires considering what the specifications will be used for, that is, the problems that humans are trying to solve when they use specifications. Previously, we looked at a narrow slice of this problem—what should be contained in blackbox requirements specifications for process control software to ensure that the resulting implementations are internally complete [JLHM91, Lev95]. This paper again considers the question of specification content, but within a larger context.

This question is critical because cognitive psychologists have determined that people tend to ignore information during problem solving that is not represented in the specification of the problem. In experiments where some problem solvers were given incomplete representations while others were not given any representation at all, those with no representation did better [FSL78, Smi89]. An incomplete problem representation actually *impaired* performance because the subjects tended to rely on it as a comprehensive and truthful representation—they failed to consider important factors deliberately omitted from the representations. Thus, being provided with an incomplete problem representation (specification) can actually lead to worse performance than having no representation at all [VR92].

One possible explanation for these results is that some problem solvers did worse because they were unaware of important omitted information. However, both novices and experts failed to use information left out of the diagrams with which they were presented, even though the experts could be expected to be aware of this information. Fischoff, who did such an experiment involving fault tree diagrams, attributed it to an "out of sight, out of mind" phenomenon [FSL78].

One place to start in deciding what should be in a system specification is with basic systems theory, which defines a *system* as a set of components that act together as a whole to achieve some common goal, objective, or end. The components are all interrelated and are either directly or indirectly connected to each other. This concept of a system relies on the assumptions that the system goals can be defined and that systems are atomistic, that is, capable of being separated into component entities such that their interactive behavior mechanisms can be described.

The system *state* at any point in time is the set of relevant properties describing the system at that time. The system *environment* is a set of components (and their properties) that are not part of the system but whose behavior can affect the system state. The existence of a boundary between the system and its environment implicitly defines as *inputs* or *outputs* anything that crosses that boundary.

It is very important to understand that a system is always a model—an abstraction *conceived by the analyst.* For the same man-made system, an observer may see a different purpose than the designer and may also focus on different relevant properties. Thus, there may be multiple "correct" system models or specifications. To ensure consistency and enhance communication, a common specification is required that defines the:

- System boundary,
- Inputs and outputs,
- Components,
- Structure,
- Relevant interactions between components and the means by which the system retains its integrity (the behavior of the components and their effect on the overall system state), and
- Purpose or goals of the system that makes it reasonable to consider it to be a coherent entity [Che81].

All of these properties need to be included in a complete system model or specification along with a description of the aspects of the environment that can affect the system state. Most of these aspects are already included in our current specification languages. However, the last, information about purpose or intent, is often not.

One of the most important limitations of the models underlying most current specification languages, both formal and informal, is that they cannot allow us to infer what is not explicitly represented in the model, including the intention of doing something a particular way. This intentional information is critical in the design and evolution of software. As Harman has said, practical reasoning is concerned with what to intend while formal reasoning with what to believe [Har82]. "Formal logic arguments are *a priori* true or false with reference to an explicitly defined model, whereas functional reasoning deals with relationships between models, and truth depends on correspondence with the state of affairs in the real world" [Har82].

In the conclusions to our paper describing our experiences specifying the requirements for TCAS II (an aircraft collision avoidance system), we wrote:

> In reverse engineering TCAS, we found it impossible to derive the requirements specification strictly from the pseudocode and an accompanying English language description. Although the basic information was all there, the intent was largely missing and often the mapping from goals or constraints to specific design decisions. Therefore, distinguishing between requirements and artifacts of the implementation was not possible in all cases. As has been discovered by most people attempting to maintain such systems, an audit trail of the decisions and the reasons why decisions were made is absolutely essential. This was not done by TCAS over the 15 years of its development, and those responsible for the system today are currently attempting to reconstruct decision-making information from old memos and corporate memory. For the most part, only one person is able to explain why some decisions were made or why things were designed in a particular way [LHHR94].

There is widespread agreement about the need for design rationale (intent) information in order to understand complex software or to correctly and efficiently change or analyze the impact of changes to it. Without a record of intent, important decisions can be undone during maintenance: Many serious accidents and losses can be traced to the fact that a system did not operate as intended because of changes that were not fully coordinated or fully analyzed to determine their effects [Lev95]. What is not so clear is the content and structure of the information that is needed.

Simply keeping an audit trail of decisions and the reasons behind them as they are made is not practical. The number of decisions made in any large project is enormous. Even if it were possible to write them all down, finding the proper information when needed seems to be a hopeless task if not structured appropriately. What is needed is a specification of the intent (goals, constraints, and design rationale) from the beginning, and it must be specified in a usable and perceptually salient manner. That is, we need a framework within which to select and specify the design decisions that are needed to develop and maintain software.

3.3 Structure

The third aspect of specifications, *structure*, is the basis for organizing information in the specification. The information may all be included somewhere, but it may be hard to find or to determine the relationship to information specified elsewhere.

Problem solving in technological systems takes place within the context of a complex causal network of relationships [Dor87, Ras86, Rea90, VR92], and those relationships need to be reflected in the specification. The information needed to solve a problem may all be included somewhere in the assorted documentation used in large projects, but it may be hard to find when needed or to determine the relationship to information specified elsewhere. Psychological experiments in problem solving find that people attend primarily to perceptually salient information [KS90]. The goal of specification language design should be to make it easy for users to extract and focus on the important information for the specific task at hand, which includes all potential tasks related to use of the specification.

Cognitive engineers speak of this problem as "information pickup" [Woo95]. Just because the information is in the interface does not mean that the operator can find it easily. The same is true for specifications. The problem of information pickup is compounded by the fact that there is so much information in system and software specifications while only a small subset of it may be relevant in any given context.

3.3.1 Complexity

The problems in building and interacting with systems correctly are rooted in complexity and intellectual manageability. A basic and often noted principle of engineering is to keep things simple. This principle, of course, is easier to state than to do. Ashby's Law of Requisite Variety [Ash62] tells us that there is a limit to how simple we can make control systems, including those designs represented in software, and still have them be effective. In addition, basic human ability is not changing. If humans want to build and operate increasingly complex systems, we need to increase what is intellectually manageable. That is, we will need to find ways to *augment* human ability.

The situation is not hopeless. As Rasmussen observes, the complexity of a system is not an objective feature of the system [Ras85]. Observed complexity depends upon the level of resolution upon which the system is being considered. A simple object becomes complex if observed through a microscope. Complexity, therefore, can only be defined with reference to a particular representation of a system, and then can only be measured relative to other systems observed at the same level of abstraction.

Thus, a way to cope with complex systems is to structure the situation such that the observer can transfer the problem being solved to a level of abstraction with less resolution. The complexity faced by the builders or

users of a system is determined by their *mental models* (representations) of the internal state of the system. We build such mental models and update them based on what we observe about the system, that is, by means of our interface to the system. Therefore, the apparent complexity of a system ultimately depends upon the technology of the interface system [Ras85].

The solution to the complexity problem is to take advantage of the most powerful resources people have for dealing with complexity. Newman has noted, "People don't mind dealing with complexity if they have some way of controlling or handling it ... If a person is allowed to structure a complex situation according to his perceptual and conceptual needs, sheer complexity is no bar to effective performance" [New66, Ras85]. Thus, complexity itself is not a problem if humans are presented with meaningful information in a coherent, structured context.

3.3.2 Hierarchy Theory

Two ways humans cope with complexity is to use top-down reasoning and stratified hierarchies. Building systems bottom-up works for relatively simple systems. But as the number of cases and objects that must be considered increases, this approach becomes unworkable—we go beyond the limits of human memory and logical ability to cope with the complexity. Top-down reasoning is a way of managing that complexity. At the same time, we have found that pure top-down reasoning is not adequate alone; humans need to combine top-down with bottom-up reasoning. Thus, the structure of the information must allow reasoning in both directions.

In addition, humans cope with complexity by building stratified hierarchies. Models of complex systems can be expressed in terms of a *hierarchy* of levels of organization, each more complex than the one below, where a level is characterized by having *emergent* properties. The concept of emergence is the idea that at any given level of complexity, some properties characteristic of that level (emergent at that level) are irreducible. Such properties do not exist at lower levels in the sense that they are meaningless in the language appropriate to those levels. For example, the shape of an apple, although eventually explainable in terms of the cells of the apple, has no meaning at that lower level of description.

Regulatory or *control* action involves imposing constraints upon the activity at one level of a hierarchy. Those constraints define the "laws of behavior" at that level that yield activity meaningful at a higher level (emergent behavior). Hierarchies are characterized by control processes operating at the interfaces between levels. Checkland explains it:

> Any description of a control process entails an upper level imposing constraints upon the lower. The upper level is a source of an alternative (simpler) description of the lower level in terms of specific functions that are emergent as a result of the imposition of constraints [Che81, pg. 87].

Hierarchy theory deals with the fundamental differences between one level of complexity and another. Its ultimate aim is to explain the relationships between different levels: what generates the levels, what separates them, and what links them. Emergent properties associated with a set of components at one level in a hierarchy are related to constraints upon the degree of freedom of those components. In the context of this paper, it is important to note that describing the emergent properties resulting from the imposition of constraints requires a language at a higher level (a metalevel) *different* than that describing the components themselves. Thus, different description languages are required at each hierarchical level.

The problem then comes down to determining appropriate types of hierarchical abstraction that allow both top-down and bottom-up reasoning. In computer science, we have made much use of part-whole abstractions where each level of a hierarchy represents an aggregation of the components at a lower level and of information-hiding abstractions where each level contains the same conceptual information but hides some details about the concepts, that is, each level is a refinement of the information at a higher level.. Each level of our software specifications can be thought of as providing *what* information while the next lower level describes *how*.

Such hierarchies, however, do not provide information about *why*. Higher-level emergent information about purpose or intent cannot be inferred from what we normally include in such specifications. Design errors may result when we either guess incorrectly about higher-level intent or omit it from our decision-making process. For example, while specifying the system requirements for TCAS II [LHHR94], we learned from experts that crossing maneuvers are avoided in the design for safety reasons. The analysis on which this decision is based comes partly from experience during TCAS system testing on real aircraft and partly as a result of an extensive safety analysis performed on the system. This design constraint would not be apparent in most design or code specifications unless it were added in the form of comments, and it could easily be violated during system modification unless it was recorded and easily located.

But there are abstractions that can be used in stratified

hierarchies other than part–whole abstraction. While investigating the design of safe human–machine interaction, Rasmussen studied protocols recorded by people working on complex systems (process plant operators and computer maintainers) and found that they structured the system along two dimensions: (1) a part–whole abstraction in which the system is viewed as a group of related components at several levels of physical aggregation, and (2) a means–ends abstraction [Ras86].

3.3.3 Means-Ends Hierarchies

In a means-end abstraction, each level represents a different model of the same system. At any point in the hierarchy, the information at one level acts as the goals (the ends) with respect to the model at the next lower level (the means). Thus, in a means–ends abstraction, the current level specifies *what*, the level below *how*, and the level above *why* [Ras86]. In essence, this intent information is emergent in the sense of system theory:

> When moving from one level to the next higher level, the change in system properties represented is not merely removal of details of information on the physical or material properties. More fundamentally, information is added on higher-level principles governing the coordination of the various functions or elements at the lower level. In man-made systems, these higher-level principles are naturally derived from the purpose of the system, i.e., from the reasons for the configurations at the level considered [Ras86]

A change of level involves both a shift in concepts and in the representation structure as well as a change in the information suitable to characterize the state of the function or operation at the various levels [Ras86].

Each level in a means-ends hierarchy describes the system in terms of a different set of attributes or "language." Models at the lower levels are related to a specific physical implementation that can serve several purposes while those at higher levels are related to a specific purpose that can be realized by several physical implementations. Changes in goals will propagate downward through the levels while changes in the physical resources (such as faults or failures) will propagate upward. In other words, states can only be described as errors or faults with reference to their intended functional purpose. Thus reasons for proper function are derived "top-down." In contrast, causes of improper function depend upon changes in the physical world (i.e., the implementation) and thus they are explained "bottom up" [VR92].

Mappings between levels are many-to-many: Components of the lower levels can serve several purposes while purposes at a higher level may be realized using several components of the lower-level model. These goal-oriented links between levels can be followed in either direction, reflecting either the means by which a function or goal can be accomplished (a link to the level below) or the goals or functions an object can affect (a link to the level above). So the means–ends hierarchy can be traversed in either a top-down (from ends to means) or a bottom-up (from means to ends) direction.

As stated earlier, our representations of problems have an important effect on our problem-solving ability and the strategies we use, and there is good reason to believe that representing the problem space as a means–ends mapping provides useful context and support for decision making and problem solving. Consideration of purpose or reason (top-down analysis in a means-ends hierarchy) has been shown to play a major role in understanding the operation of complex systems [Ras85].

Rubin's analysis of his attempts to understand the function of a camera's shutter (as cited in [Ras90]) provides an example of the role of intent or purpose in understanding a system. Rubin describes his mental efforts in terms of conceiving all the elements of the shutter in terms of their function in the whole rather than explaining how the individual parts worked: How they worked was immediately clear when their function was known. Rasmussen argues that this approach has the advantage that solutions of subproblems are identifiable with respect to their place in the whole picture, and it is immediately possible to judge whether a solution is correct or not. In contrast, arguing from the parts to the way they work is much more difficult because it requires synthesis: Solutions of subproblems must be remembered in isolation, and their correctness is not immediately apparent.

Support for this argument can be found in the difficulties AI researchers have encountered when modeling the function of mechanical devices "bottom-up" from the function of the components. DeKleer and Brown found that determining the function of an electric buzzer solely from the structure and behavior of the parts requires complex reasoning [DB83]. Rasmussen suggests that the resulting inference process is very artificial compared to the top-down inference process guided by functional considerations as described by Ruben. "In the DeKleer-Brown model, it will be difficult to see the woods for the trees, while Rubin's description appears to be guided by a birds-eye perspective " [Ras90].

Glaser and Chi suggest that experts and successful problem solvers tend to focus first on analyzing the functional structure of the problem at a high level of ab-

straction and then narrow their search for a solution by focusing on more concrete details [GC88]. Representations that constrain search in a way that is explicitly related to the purpose or intent for which the system is designed have been shown to be more effective than those that do not because they facilitate the type of goal-directed behavior exhibited by experts [VCP95]. Therefore, we should be able to improve the problem solving required in software development and evolution tasks by providing a representation (i.e., specification) of the system that facilitates goal-oriented search by making explicit the goals related to each component.

Viewing a system from a high level of abstraction is not limited to a means–ends hierarchy, of course. Most hierarchies allow one to observe systems at a less detailed level. The difference is that the means–ends hierarchy is explicitly *goal oriented* and thus assists goal-oriented problem solving. With other hierarchies (such as the part–whole hierarchies often used in computer science), the links between levels are not necessarily related to goals. So although it is possible to use higher-levels of abstraction to select a subsystem of interest and to constrain search, the subtree of the hierarchy connected to a particular subsystem does not necessarily contain system components relevant to the goals the problem solver is considering.

3.4 Form (Notation)

The final aspect of specification design is the actual form of the specification. Although this is often where we start when designing languages, the four aspects actually have to be examined in order, first defining the process to be supported, then determining what the content should be, then how the content will be structured to make the information easily located and used, and finally the form the language should take. All four aspects need to be addressed not only in terms of the analysis to be performed on the specification, but also with respect to human perceptual and cognitive capabilities.

Note that the form itself must also be considered from a psychological standpoint: The usability of the language will depend on human perceptual and cognitive strategies. For example, Fitter and Green describe the attributes of a good notation with respect to human perception and understanding [FG79]. Casner [Cas91] and others have argued that the utility of any information presentation is a function of the *task* that the presentation is being used to support. For example, a symbolic representation might be better than a graphic for a particular task, but worse for others.

No particular specification language is being proposed here. We first must clarify what needs to be expressed before we can design languages that express that information appropriately and effectively. In addition, different types of systems require different types of languages. All specifications are abstractions—they leave out unimportant details. What is important will depend on the problem being solved. For different types of systems, the important and difficult aspects differ. For example, specifications for embedded controllers may emphasize control flow over data flow (which is relatively trivial for these systems), while data transformation or information management systems might place more emphasis on the specification of data flow than control flow. Attempts to include everything in the specification are not only impractical, but involve wasted effort and are unlikely to fit the budgets and schedules of industry projects. Because of the importance of completeness, as argued earlier, determining exactly what needs to be included becomes the most important problem in specification design.

This paper deals with process, content and structure, but not form (notation). We are defining specification languages built upon the foundation laid in this paper and on other psychological principles, but they will be described in future papers.

4 Intent Specifications

These basic ideas provide the foundation for what I call *intent specifications*. They have been developed and used successfully in cognitive engineering by Vicente and Rasmussen for the design of operator interfaces, a process they call *ecological interface design* [DV96, Vic91].

The exact number and content of the means–ends hierarchy levels may differ from domain to domain. Here I present a structure for process systems with shared software and human control. In order to determine the feasibility of this approach for specifying a complex system, I extended the formal TCAS II aircraft collision avoidance system requirements specification we previously wrote [LHHR94] to include intent information and other information that cannot be expressed formally but is needed in a complete system requirements specification. We are currently applying the approach to other examples, including a NASA robot and part of the U.S. Air Traffic Control System. The TCAS II specification is used as an example in this paper.[2] The table of contents for the example TCAS II System Requirements Specification (shown in Figure 3) may be helpful in fol-

[2]Our TCAS II Intent Specification (complete system specification) is over 800 pages long. Obviously, the entire specification cannot be included in this paper. It can be accessed from http://www.cs.washington.edu/homes/leveson.

lowing the description of intent specifications. Note that the only part of TCAS that we specified previously is section 3.4 and parts of 3.3.

In the intent specifications we have built for real systems, we have found the approach to be practical; in fact, most of the information in an intent specification is already located somewhere in the often voluminous documentation for large systems. The problem in these systems usually lies in finding specific information when it is needed, in tracing the relationships between information, and in understanding the system design and why it was designed that way. Intent specifications are meant to assist with these tasks.

System and software specifications of the type being proposed, like those used in ecological interface design, are organized along two dimensions: intent abstraction and part-whole abstraction (see Figure 2). These two dimensions constitute the problem space in which the human navigates. The part-whole (horizontal) dimension, which itself can be separated into refinement and decomposition, allows users to change their focus of attention to more or less detailed views within each level or model. The vertical dimension specifies the level of intent at which the problem is being considered, i.e., the language or model that is currently being used.

4.1 Part-Whole Dimension

Computer science commonly uses two types of part–whole abstractions. *Parallel decomposition* (or its opposite, aggregation) separates units into (perhaps interacting) components of the same type. In Statecharts, for example, these components are called orthogonal components and the process of aggregation results in an orthogonal product. Each of the pieces of the parallel decomposition of Statecharts is a state machine, although each state machine will in general be different.

The second type of part-whole abstraction—*refinement*—takes a function and breaks it down into more detailed steps. An example is the combining of a set of states into a superstate in Statecharts. In Petrinets, such abstractions have been applied both to states and to transitions—they provide a higher-level name for a piece of the net. In programming, refinement abstractions are represented by procedures or subprograms.

Note that neither of these types of abstraction is an "emergent-property" or means–ends abstraction—the whole is simply broken up into a more detailed description. Additional information, such as intent, is not provided at the higher level.

Along this horizontal dimension, intent specifications are broken up into three parts. The first column contains information about characteristics of the environment that affects the ability to achieve the system goals and design constraints. For example, in TCAS, the designers need information about the operation of the ground-based ATC system in order to fulfill the system-level constraint of not interfering with it. Information about the environment is also needed for some types of hazard analysis and for normal system design. For example, the design of the surveillance logic in TCAS depends on the characteristics of the transponders carried on the aircraft with which the surveillance logic interacts.

The second column of the horizontal dimension is information about human operators or users. Too often human factors design and software design is done independently. Many accidents and incidents in aircraft with advanced automation have been blamed on human-error that has been induced by the design of the automation. For example, Weiner introduced the term *clumsy automation* to describe automation that places additional and unevenly distributed workload, communication, and coordination demands on pilots without adequate support [Wei89]). Sarter, Woods, and Billings [SWB95] describe additional problems associated with new attentional and knowledge demands and breakdowns in mode awareness and "automation surprises," which they attribute to *technology-centered automation*: Too often, the designers of the automation focus exclusively on technical aspects, such as the mapping from software inputs to outputs, on mathematical models of requirements functionality, and on the technical details and problems internal to the computer; they do not devote enough attention to the cognitive and other demands of the automation design on the operator.

One goal of intent specifications is to integrate the information needed to design "human-centered automation" into the system requirements specification. We are also working on analysis techniques to identify problematic system and software design features in order to predict where human errors are likely to occur [LPS97]. This information can be used in both the automation design and in the design of the operator procedures, tasks, interface, and training.

The third part of the horizontal dimension is the system itself and its decomposition along the part–whole dimension.

4.2 Intent Dimension

The Intent (vertical) dimension has five hierarchical levels, each providing intent ("why") information about the level below. Each level is mapped to the appropriate parts of the intent levels above and below it, providing

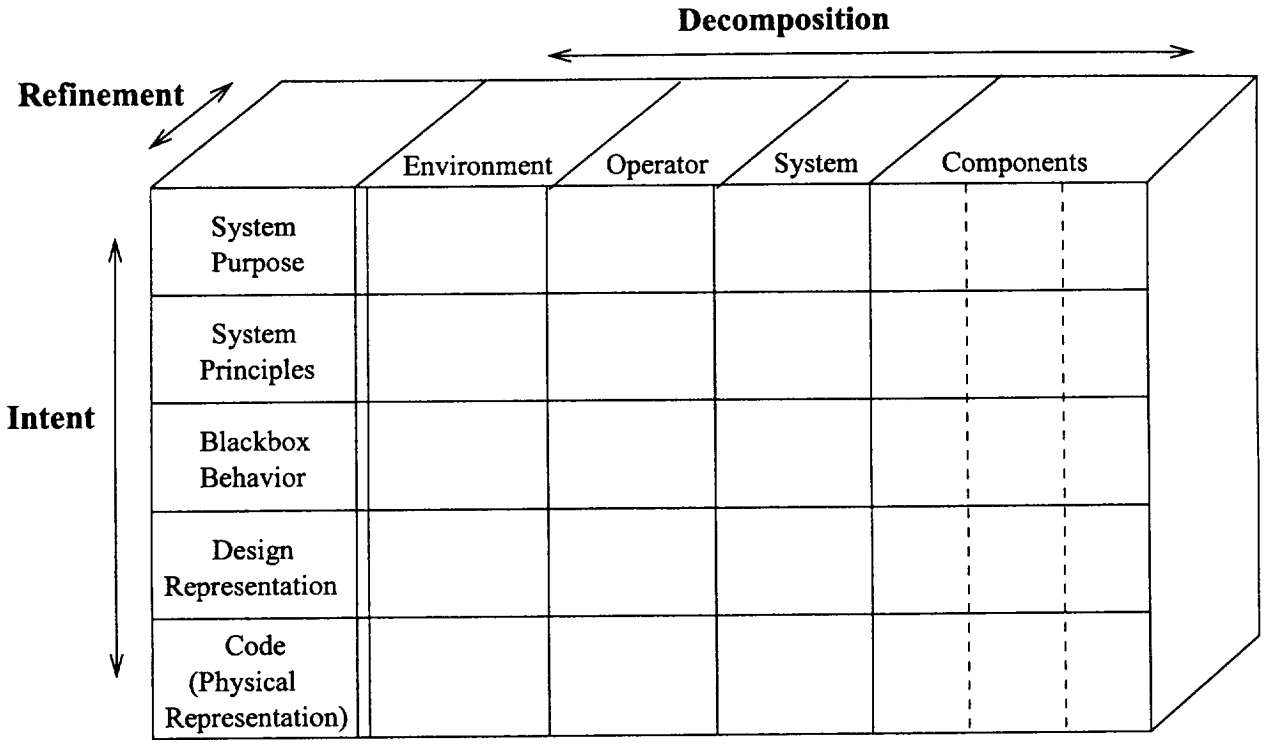


Figure 2: The structure of an intent specification for software systems.

traceability of high-level system requirements and constraints down to code (or physical form) and vice versa.

Each level also supports a different type of reasoning about the system, with the highest level assisting systems engineers in their reasoning about system-level goals, constraints, priorities, and tradeoffs. The second level, System Design Principles, allows engineers to reason about the system in terms of the physical principles and laws upon which the design is based. The Blackbox Behavior level enhances reasoning about the logical design of the system as a whole and the interactions between the components as well as the functional state without being distracted by implementation issues. The lowest two levels provide the information necessary to reason about individual component design and implementation issues. The mappings between levels provide the relational information that allows reasoning across hierarchical levels.

Each level (except the top level) also includes a specification of the requirements and results of verification or validation activities for the information at that specification level. The top level does not include this information (except perhaps for parts of the hazard analysis) because it is not clear what types of validation, outside of expert review, would be appropriate at this highest level of intent abstraction.

4.2.1 System Purpose

Along the vertical dimension, the highest specification level, *System Purpose*, contains the (1) system goals, (2) design constraints, (3) assumptions, (4) limitations, (5) design evaluation criteria and priorities, and (6) results of analyses for system level qualities.

Examples of high-level *goals* (purpose) for TCAS II are to:

G1: *Provide affordable and compatible collision avoidance system options for a broad spectrum of National Airspace System users.*

G2: *Detect potential midair collisions with other aircraft in all meteorological conditions.*

Usually, in the early stages of a project, goals are stated in very general terms. One of the first steps in defining system requirements is to refine the goals into testable and achievable high-level requirements. For G1 above, a refined subgoal is:

R1: *Provide collision avoidance protection for any two aircraft closing horizontally at any rate up to 1200 knots and vertically up to 10,000 feet per minute.*

This type of refinement and reasoning is done at the System Purpose level, using an appropriate specification language (most likely English).

Requirements (and constraints) are also included for the

1. System Purpose

- 1.1 Introduction
- 1.2 Historical Perspective
- 1.3 Environment
  - 1.3.1 Environmental Assumptions
  - 1.3.2 Environmental Constraints
- 1.4 Operator
  - 1.4.1 Tasks and Procedures
  - 1.4.2 Pilot-TCAS Interface Requirements
- 1.5 TCAS System Goals
- 1.6 High-Level Functional Requirements
- 1.7 System Limitations
- 1.8 System Constraints
  - 1.8.1 General Constraints
  - 1.8.2 Safety-Related Constraints
- 1.9 Hazard Analysis

2. System Design Principles

- 2.1 General Description
- 2.2 TCAS System Components
- 2.3 Surveillance and Collision Avoidance Logic
  - 2.3.1 General Concepts
  - 2.3.2 Surveillance
  - 2.3.3 Tracking
  - 2.3.4 Traffic Advisories
  - 2.3.5 Resolution Advisories
  - 2.3.6 TCAS/TCAS Coordination
- 2.4 Performance Monitoring
- 2.5 Pilot-TCAS Interface
  - 2.5.1 Controls
  - 2.5.2 Displays and Aural Annunciations
- 2.6 Testing and Validation
  - 2.6.1 Simulations
  - 2.6.2 Experiments
  - 2.6.3 Other Validation Procedures and Results

3. Blackbox Behavior

- 3.1 Environment
- 3.2 Flight Crew Requirements
  - 3.2.1 Tasks
  - 3.2.2 Operational Procedures
- 3.3 Communication and Interfaces
  - 3.3.1 Pilot-TCAS Interface
  - 3.3.2 Message Formats
  - 3.3.3 Input Interfaces
  - 3.3.4 Output Interfaces
  - 3.3.5 Receiver, Transmitter, Antennas
- 3.4 Behavioral Requirements
  - 3.4.1 Surveillance
  - 3.4.2 Collision Avoidance
  - 3.4.3 Performance Monitoring
- 3.5 Testing Requirements

4. Physical and Logical Function

- 4.1 Human-Computer Interface Design
- 4.2 Pilot Operations (Flight) Manual
- 4.3 Software Design
- 4.4 Physical Requirements
  - 4.4.1 Definition of Standard Conditions
  - 4.4.2 Performance Capability of Own Aircraft's Mode S Transponder
  - 4.4.3 Receiver Characteristics
  - 4.4.4 TCAS Transmitter Characteristics
  - 4.4.5 TCAS Transmitter Pulse Characteristics
  - 4.4.6 TCAS Pulse Decoder Characteristics
  - 4.4.7 Interfence Limiting
  - 4.4.8 Aircraft Suppression Bus
  - 4.4.9 TCAS Data Handling and Interfaces
  - 4.4.10 Bearing Estimation
  - 4.4.11 High-Density Techniques
- 4.5 Hardware Design Specifications
- 4.6 Verification Requirements

5. Physical Realization

- 5.1 Software
- 5.2 Hardware Assembly Instructions
- 5.3 Training Requirements (Plan)
- 5.4 Maintenance Requirements

A. Constant Definitions

B. Table Definitions

C. Reference Algorithms

D. Physical Measurement Conventions

E. Performance Requirements on Equipment that Interacts with TCAS

F. Glossary

G. Notation Guide

H. Index

Figure 3: The contents of the sample TCAS Intent Specification

human operator, for the human-computer interface, and for the environment in which TCAS will operate. Requirements on the operator (in this case, the pilot) are used to guide the design of the TCAS-pilot interface, flightcrew tasks and procedures, aircraft flight manuals, and training plans and program. Links are provided to show the relationships. Example TCAS II operator requirements are:

O1: *After the threat is resolved, the pilot shall return promptly and smoothly to his/her previously assigned flight path.*

O2: *The pilot must not maneuver on the basis of a Traffic Advisory only.*

Design constraints are restrictions on how the system can achieve its purpose. For example, TCAS is not allowed to interfere with the ground-level air traffic control system while it is trying to maintain adequate separation between aircraft. Avoiding interference is not a goal or purpose of TCAS—the best way to achieve it is not to build the system at all. It is instead a constraint on how the system can achieve its purpose, i.e., a constraint on the potential system designs. Because of the need to evaluate and clarify tradeoffs among alternative designs, separating these two types of intent information (goals and design constraints) is important.

For safety-critical systems, constraints should be further separated into normal and safety-related. Examples of *non-safety constraints* for TCAS II are:

C1: *The system must use the transponders routinely carried by aircraft for ground ATC purposes.*

C2: *No deviations from current FAA policies and philosophies must be required.*

Safety-related constraints should have two-way links to the system hazard log and perhaps links to any analysis results that led to that constraint being identified. Hazard analyses specified on this level are linked to Level 1 requirements and constraints on this level, to design features on Level 2, and to system limitations (or accepted risks). Example safety constraints are:

SC1: *The system must generate advisories that require as little deviation as possible from ATC clearances.*

SC2: *The system must not disrupt the pilot and ATC operations during critical phases of flight.*

Note that *refinement* occurs at the same level of the intent specification (see Figure 2). For example, the safety-constraint SC3 can be refined

SC1: *The system must not interfere with the ground ATC system or other aircraft transmissions to the ground ATC system.*

SC1.1: *The system design must limit interference with ground-based secondary surveillance radar, distance-measuring equipment channels, and with other radio services that operate in the 1030/1090 MHz frequency band.*

SC1.1.1: *The design of the Mode S waveforms used by TCAS must provide compatibility with Modes A and C of the ground-based secondary surveillance radar system.*

SC1.1.1: *The frequency spectrum of Mode S transmissions must be controlled to protect adjacent distance-measuring equipment channels.*

SC1.1.1: *The design must ensure electromagnetic compatibility between TCAS and* ....

SC1.2: *Multiple TCAS units within detection range of one another (approximately 30 nmi) must be designed to limit their own transmissions. As the number of such TCAS units within this region increases, the interrogation rate and power allocation for each of them must decrease in order to prevent undesired interference with ATC.*

Environment requirements and constraints may lead to restrictions on the use of the system or to the need for system safety and other analyses to determine that the requirements hold for the larger system in which the system being designed is to be used. Examples for TCAS include:

E1: *Among the aircraft environmental alerts, the hierarchy shall be: Windshear has first priority, then the Ground Proximity Warning System (GPWS), then TCAS.*

E2: *The behavior or interaction of non-TCAS equipment with TCAS must not degrade the performance of the TCAS equipment or the performance of the equipment with which TCAS interacts.*

E3: *The TCAS alerts and advisories must be independent of those using the master caution and warning system.*

Assumptions are specified, when appropriate, at all levels of the intent specification to explain a decision or to record fundamental information on which the design is based. These assumptions are often used in the safety or other analyses or in making lower level design decisions. For example, operational safety depends on the accuracy of the assumptions and models underlying the design and hazard analysis processes. The operational system should be monitored to ensure (1) that it is constructed, operated, and maintained in the manner assumed by the designers, (2) that the models and assumptions used during initial decision making and design were correct, and (3) that the models and assumptions are not violated by changes in the system,

such as workarounds or unauthorized changes in procedures, or by changes in the environment [Lev95]. Operational feedback on trends, incidents, and accidents should trigger reanalysis when appropriate. Linking the assumptions throughout the document with the hazard analysis (for example, to particular boxes in the system fault trees) will assist in performing safety maintenance activities.

Examples of assumptions associated with requirements on the first level of the TCAS intent specification:

R1: *Provide collision avoidance protection for any two aircraft closing horizontally at any rate up to 1200 knots and vertically up to 10,000 feet per minute.*

> **Assumption**: *This requirement is derived from the assumption that commercial aircraft can operate up to 600 knots and 5000 fpm during vertical climb or controlled descent (and therefore two planes can close horizontally up to 1200 knots and vertically up to 10,000 fpm).*

R3: *TCAS shall operate in enroute and terminal areas with traffic densities up to 0.3 aircraft per square nautical miles (i.e., 24 aircraft within 5 nmi).*

> **Assumption**: *Traffic density may increase to this level by 1990, and this will be the maximum density over the next 20 years.*

An example of an assumption associated with a safety constraint is:

SC5: *The system must not disrupt the pilot and ATC operations during critical phases of flight nor disrupt aircraft operation.*

SC5.1: *The pilot of a TCAS-equipped aircraft must have the option to switch to the Traffic-Advisory-Only mode where TAs are displayed but display of resolution advisories is inhibited.*

> **Assumption**: *This feature will be used during final approach to parallel runways, when two aircraft are projected to come close to each other and TCAS would call for an evasive maneuver.*

Assumptions may also apply to features of the environment. Examples of environment assumptions for TCAS are that:

EA1: *All aircraft have legal identification numbers.*
EA2: *All aircraft carry transponders.*
EA3: *The TCAS-equipped aircraft carries a Mode-S air traffic control transponder, whose replies include encoded altitude when appropriately interrogated.*
EA4: *Altitude information is available from intruding targets with a minimum precision of 100 feet.*
EA5: *Threat aircraft will not make an abrupt maneuver that thwarts the TCAS escape maneuver.*

System *limitations* are also specified at Level 1 of an intent specification. Some may be related to the basic functional requirements, such as:

L1: *TCAS does not currently indicate horizontal escape maneuvers and therefore does not (and is not intended to) increase horizontal separation.*

Limitations may also relate to environment assumptions. For example, system limitations related to the environment assumptions above include:

L2: *TCAS provides no protection against aircraft with nonoperational transponders.*
L3: *Aircraft performance limitations constrain the magnitude of the escape maneuver that the flight crew can safely execute in response to a resolution advisory. It is possible for these limitations to preclude a successful resolution of the conflict.*
L4: *TCAS is dependent on the accuracy of the threat aircraft's reported altitude. Separation assurance may be degraded by errors in intruder pressure altitude as reported by the transponder of the intruder aircraft.*

> **Assumption:** *This limitation holds for existing airspace, where many aircraft use pressure altimeters rather than GPS. As more aircraft install GPS systems with greater accuracy than current pressure altimeters, this limitation will be reduced or eliminated.*

Limitations are often associated with hazards or hazard causal factors that could not be completely eliminated or controlled in the design. Thus they represent accepted risks. For example:

L5: *TCAS will not issue an advisory if it is turned on or enabled to issue resolution advisories in the middle of a conflict (→FTA-405)*[3].
L6: *If only one of two aircraft is TCAS equipped while the other has only ATCRBS altitude-reporting capability, the assurance of safe separation may be reduced (→FTA-290).*

In our TCAS intent specification, both of these system limitations have pointers to boxes in the fault tree generated during the hazard analysis of TCAS II.

Finally, limitations may be related to problems encountered or tradeoffs made during the system design process (recorded on lower levels of the intent specification). For example, TCAS has a Level 1 performance monitoring requirement that led to the inclusion of a self-test function in the system design to determine whether TCAS

[3]The pointer to FTA-405 denotes the box labelled 405 in the Level-1 fault tree analysis

is operating correctly. The following system limitation relates to this self-test facility:

L7: *Use by the pilot of the self-test function in flight will inhibit TCAS operation for up to 20 seconds depending upon the number of targets being tracked. The ATC transponder will not function during some portion of the self-test sequence.*

Most of these system limitations will be traced down in the intent specification levels to the user documentation. In the case of an avionics system like TCAS, this specification includes the Pilot Operations (Flight) Manual on level 4 of our TCAS intent specification. An example is shown in the next section.

Evaluation criteria and *priorities* are used to resolve conflicts among goals and design constraints and to guide design choices at lower levels. This information has not been included in the TCAS example specification as I was unable to find out how these decisions were made during the TCAS design process.

Finally, Level 1 contains the analysis results for system-level (emergent) properties such as safety or security. For the TCAS specification, a hazard analysis (including fault tree analysis and failure modes and effects analysis) was performed and is included and linked to the safety-critical design constraints on this level and to lower-level design decisions based on the hazard analysis. Whenever changes are made in safety-critical systems or software (during development or during maintenance and evolution), the safety of the change needs to be evaluated. This process can be difficult and expensive. By providing links throughout the levels of the intent specification, it should be easy to assess whether a particular design decision or piece of code was based on the original safety analysis or safety-related design constraint.

4.2.2 System Design Principles

The second level of the specification contains *System Design Principles*—the basic system design and scientific and engineering principles needed to achieve the behavior specified in the top level. The horizontal dimension again allows abstraction and refinement of the basic system principles upon which the design is predicated.

For TCAS, this level includes such general principles as the basic *tau* concept, which is related to all the high-level alerting goals and constraints:

PR1: *Each TCAS-equipped aircraft is surrounded by a protected volume of airspace. The boundaries of this volume are shaped by the tau and DMOD criteria.*

PR1.1: *TAU: In collision avoidance, time-to-go to the closest point of approach (CPA) is more important than distance-to-go to the CPA. Tau is an approximation of the time in seconds to CPA. Tau equals 3600 times the slant range in nmi, divided by the closing speed in knots.*

PR1.2: *DMOD: If the rate of closure is very low, a target could slip in very close without crossing the tau boundaries and triggering an advisory. In order to provide added protection against a possible maneuver or speed change by either aircraft, the tau boundaries are modified (called DMOD). DMOD varies depending on own aircraft's altitude regime. See Table 2.*

The principles are linked to the related higher level requirements, constraints, assumptions, limitations, and hazard analysis as well as linked to lower-level system design and documentation. Assumptions used in the formulation of the design principles may also be specified at this level. For example, the TCAS design has a built-in bias against generating advisories that would result in the aircraft crossing paths (called *altitude crossing advisories*).

PR36.2: *A bias against altitude crossing RAs is also used in situations involving intruder level-offs at least 600 feet above or below the TCAS aircraft. In such a situation, an altitude-crossing advisory is deferred if an intruder aircraft that is projected to cross own aircraft's altitude is more than 600 feet away vertically* ($\downarrow Alt\_Separation\_Test_{m-351}$).

> **Assumption:** *In most cases, the intruder will begin a level-off maneuver when it is more than 600 feet away and so should have a greatly reduced vertical rate by the time it is within 200 feet of its altitude clearance (thereby either not requiring an RA if it levels off more than* ZTHR[4] *feet away or requiring a non-crossing advisory for level-offs begun after* ZTHR *is crossed but before the 600 foot threshold is reached).*

The example above includes a pointer down to the part of the black box requirements specification (*Alt_Separation_Test*) that embodies the design principle. As another example of the type of links that may be found between Level 2 and the levels above and below it, consider the following. TCAS II advisories may need to be inhibited because of an inadequate climb performance for the particular aircraft on which TCAS II is installed. The collision avoidance maneuvers posted

[4]The vertical dimension, called ZTHR, used to determine whether advisories should be issued varies from 750 to 950 feet, depending on the TCAS aircraft's altitude.

as advisories (called RAs or Resolution Advisories) by TCAS II assume an aircraft's ability to safely achieve them. If it is likely they are beyond the capability of the aircraft, then TCAS II must know beforehand so it can change its strategy and issue an alternative advisory. The performance characteristics are provided to TCAS II through the aircraft interface. An example design principle (related to this problem) found on Level 2 of the intent specification is:

PR39: *Because of the limited number of inputs to TCAS for aircraft performance inhibits, in some instances where inhibiting RAs would be appropriate it is not possible to do so (↑L3). In these cases, TCAS may command maneuvers that may significantly reduce stall margins or result in stall warning (↑SC9.1). Conditions where this may occur include .... The aircraft flight manual or flight manual supplement should provide information concerning this aspect of TCAS so that flight crews may take appropriate action (↓ [Pilot procedures on Level 3 and Aircraft Flight Manual on Level 4).*

Finally, principles may reflect tradeoffs between higher-level goals and constraints. As examples:

PR2: *Tradeoffs must be made between necessary protection (G1) and unnecessary advisories (SC5). This is accomplished by controlling the sensitivity level, which controls the tau, and therefore the dimensions of the protected airspace around each TCAS-equipped aircraft. The greater the sensitivity level, the more protection is provided but the higher is the incidence of unnecessary alerts. Sensitivity level is determined by ...*

PR38: *The need to inhibit* CLIMB *RAs because of inadequate aircraft climb performance will increase the likelihood of TCAS II (a) issuing crossing maneuvers, which in turn increases the possibility that an RA may be thwarted by the intruder maneuvering (↑SC7.1, FTA-1150), (b) causing an increase in* DESCEND *RAs at low altitude (↑SC8.1), and (c) providing no RAs if below the descend inhibit level (1200 feet above ground level on takeoff and 1000 feet above ground level on approach).*

4.2.3 Blackbox Behavior

Beginning at the third level or *Blackbox Behavior* level, the specification starts to contain information more familiar to software engineers. Above this level, much of the information, if located anywhere, is found in system engineering specifications. The Blackbox Behavior model at the whole system viewpoint specifies the system components and their interfaces, including the human components (operators). Figure 4 shows a system-level view of TCAS II and its environment. Each system component behavioral description and each interface is refined in the normal way along the horizontal dimensions.

The environment description includes the assumed behavior of the external components (such as the altimeters and transponders for TCAS), including perhaps failure behavior, upon which the correctness of the system design is predicated, along with a description of the interfaces between the TCAS system and its environment. Figure 5 shows part of a state-machine description of an environment component, in this case an altimeter.

Remember that the boundaries of a system are purely an abstraction and can be set anywhere convenient for the purposes of the specifier. In this case, I included as environment any component that was already on the aircraft or in the airspace control system and was not newly designed or built as part of the TCAS effort.

Going along this level to the right, each arrow in Figure 4 represents a communication and needs to be described in more detail. Each box (component) also needs to be refined. What is included in the decomposition of the component will depend on whether the component is part of the environment or part of the system being constructed. The language used to describe the components may also vary. I use a state-machine language called SpecTRM-RL (Specification Tools and Requirements Methodology–Requirements Language), which is a successor to the language (RSML) used in our official TCAS II specification [LHHR94]. Figure 6 shows part of the SpecTRM-RL description of the behavior of the CAS (collision avoidance system) subcomponent. SpecTRM-RL specifications are intended to be both easily readable with minimum instruction and formally analyzable (we have a set of analysis tools that work on these specifications).

Note that the behavioral descriptions at this level are purely blackbox: They describe the inputs and outputs of each component and their relationships *only* in terms of externally visible variables, objects, and mathematical functions. Any of these components (except the humans, of course) could be implemented either in hardware or software (and, in fact, some of the TCAS surveillance functions are implemented using analog devices by some vendors). Decisions about physical implementation, software design, internal variables, and so on are limited to levels of the specification below this one.

Other information at this level might include flight crew requirements such as description of tasks and operational procedures, interface requirements, and the testing requirements for the functionality described on this level. We have developed a visual operator task modeling language that can be translated to SpecTRM-RL

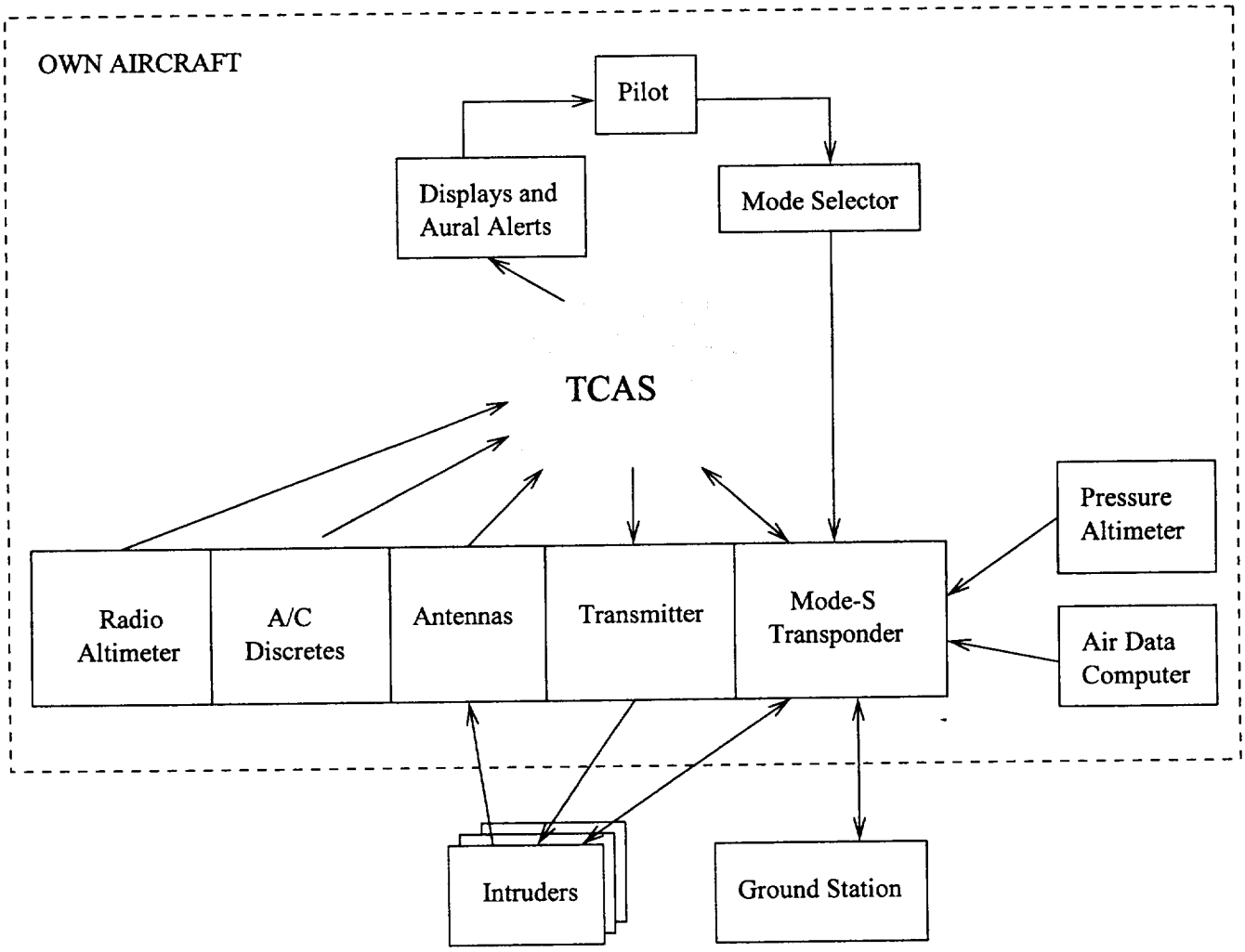


Figure 4: System viewpoint showing the system interface topology for the Blackbox Behavior level of the TCAS specification.

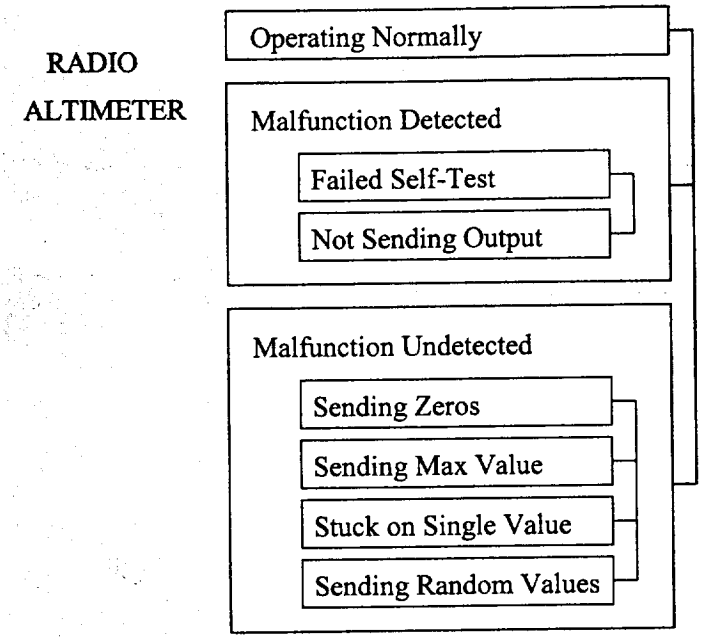


Figure 5: Part of the SpecTRM-RL description of an environment component (a radio altimeter). Modeling failure behavior is especially important for safety analyses. In this example, (1) the altimeter may be operating correctly, (2) it may have failed in a way that the failure can be detected by TCAS II (i.e., it fails a self-test and sends a status message to TCAS or it is not sending any output at all), or (3) the malfunctioning is undetected and it sends an incorrect radio altitude.

and thus permits integrated simulation and analysis of the entire system, including human-computer interactions [BL98].

4.2.4 Design Representation

The two lowest levels of an intent specification provide the information necessary to reason about component design and implementation. The fourth level, *Design Representation*, contains design information. Its content will depend on whether the particular function is being implemented using analog or digital devices or both. In any case, this level is the first place where the specification should include information about the physical or logical implementation of the components.

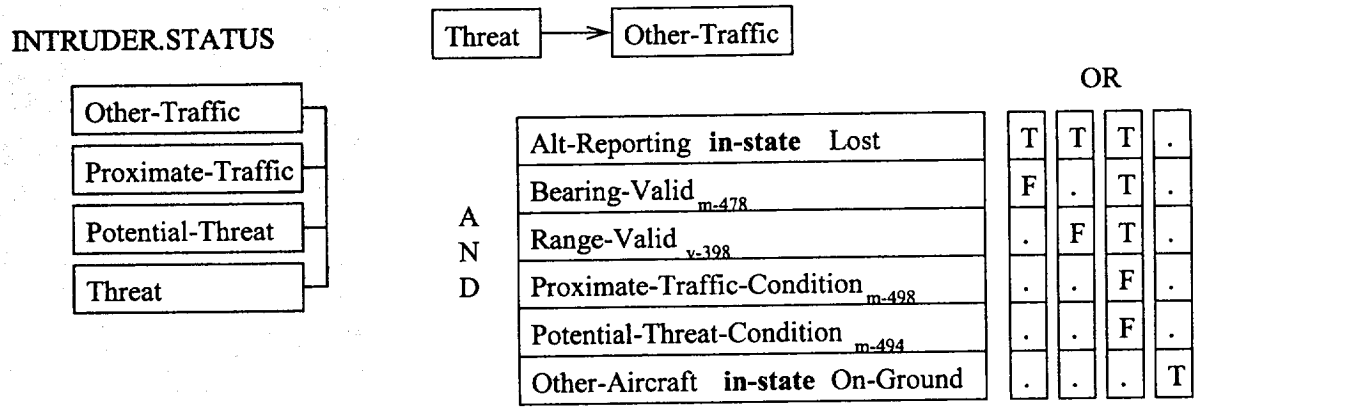
For functions implemented on digital computers, the fourth level might contain the usual software design documents or it might contain information different from that normally specified. Again, this level is linked to the higher level specification.

The design intent information may not all be completely linked and traceable upward to the levels above the Design Representation—for example, design decisions based on performance or other issues unrelated to requirements or constraints, such as the use of a particular graphics package because the programmers are familiar with it or it is easy to learn. Knowing that these decisions are *not* linked to higher level purpose is important during software maintenance and evolution activities.

The fourth level of the example TCAS intent specification simply contains the official pseudocode design specification. But this level might contain information different than we usually include in design specifications. For example, Soloway et.al. [Sol88] describe the problem of modifying code containing delocalized plans (plans or schemas with pieces spread throughout the software). They recommend using pointers to chain the pieces together, but a more effective approach might be to put the plan or schema at the higher design representation level and point to the localized pieces in the lower level Code or Physical representation. The practicality of this approach, of course, needs to be determined.

Soloway et.al. also note that reviewers have difficulty reviewing and understanding code that has been optimized. To assist in code reviews and walkthroughs, the unoptimized code sections might be shown in the refinement of the Design Representation along with mappings to the actual optimized code at the lower implementation level.

The possibilities for new types of information and rep-



Description: A threat is reclassified as other traffic if its altitude reporting has been lost (⩓PR13) and either the bearing or range inputs are invalid; if its altitude reporting has been lost and both the range and bearing are valid but neither the proximate nor potential threat classification criteria are satisfied; or the aircraft is on the ground (⩓PR12).

Mapping to Level 2: ⩓PR23, ⩓PR29

Mapping to Level 4: ⩔Section 7.1, Traffic-Advisory

Figure 6: Part of a SpecTRM-RL Blackbox Behavior level description of the criteria for downgrading the status of an intruder (into our protected volume) from being labeled a threat to being considered simply as other traffic. Intruders can be classified in decreasing order of importance as a threat, a potential threat, proximate traffic, and other traffic. In the example, the criterion for taking the transition from state *Threat* to state *Other Traffic* is represented by an AND/OR table, which evaluates to TRUE if any of its columns evaluates to TRUE. A column is TRUE if all of its rows that have a "T" are TRUE and all of its rows with an "F" are FALSE. Rows containing a dot represent "don't care" conditions. The subscripts denote the type of expression (e.g., v for input variable, m for macro, t for table, and f for function) as well as the page in the document on which the expression is defined. A macro is simply an AND/OR table used to implement an abstraction that simplifies another table.

resentations at this level of the intent hierarchy is the subject of long-term research.

Other information at this level might include hardware design descriptions, the human–computer interface design specification, the pilot operations (flight) manual, and verification requirements for the requirements and design specified on this level.

4.2.5 Physical Representation

The lowest level includes a description of the physical implementation of the levels above. It might include the software itself, hardware assembly instructions, training requirements (plan), etc.

4.2.6 Example

To illustrate this approach to structuring specifications, a small example is used related to generating resolution advisories. TCAS selects a resolution advisory (vertical escape maneuver) against other aircraft that are considered a threat to the aircraft on which the TCAS system resides. A resolution advisory (RA) has both a sense (upward or downward) and a strength (vertical rate), and it can be positive (e.g., CLIMB) or negative (e.g., DON'T CLIMB). In the software to evaluate the sense to be chosen against a particular threat, there is a procedure to compute what is called a "Don't-Care-Test." The software itself (Level 5) would contain comments about implementation decisions and also a pointer up to the Level 4 design documentation and from there up to the Level 3 black-box description of this test, shown in Figure 7.

In turn, the blackbox (Level 3) description of the Dont-Care-Test would be linked to Level 2 explanations of the intent of the test and the reason behind (why) the design of the test. For example, our Level 2 TCAS intent specification contains the following:

> **PR35: Don't-Care-Test.** *When TCAS is displaying an RA against one threat and then attempts to choose a sense against a second threat, it is often desirable to choose the same sense against it as was chosen against the first threat, even if this sense is not optimal for the new threat. One advantage is display continuity (↑ SC6). Another advantage is that the pilot may maneuver more sharply to increase separation against both threats. If a dual sense advisory is given, such as* DON'T CLIMB AND DON'T DESCEND, *a vertical maneuver to increase separation against one threat reduces separation against the other threat. The most important advantage, however, is to avoid sacrificing separation inappropriately against the first threat in order to gain a marginal advantage against the second threat.*
>
> *The don't-care test determines the relative advantages of optimizing the sense against the new threat versus selecting the same sense for both threats. When the former outweighs the latter, the threat is called a do-care threat; otherwise, the threat is a don't-care threat.*

This Level 2 description in turn points up to high-level goals to maintain separation between aircraft and constraints (both safety-related and non-safety-related) on how this can be achieved. We found while constructing the TCAS intent specification that having to provide these links identified goals and constraints that did not seem to be documented anywhere but were implied by the design and some of the design documentation.

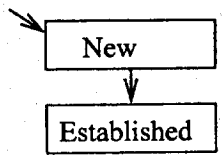
Understanding the design of the Don't-Care-Test also requires understanding other concepts of sense selection and aircraft separation requirements that are used in the blackbox description (and in the implementation) of the Don't-Care-Test procedure. For example, the separation between aircraft in Figure 7 is defined in terms of ALIM. The concept is used in the Level 3 documentation, but the meaning and intent behind using the concept is defined in the basic TCAS design principles at Level 2:

> **PR2: ALIM.** ALIM *is the desired or "adequate" amount of separation between aircraft that TCAS is designed to meet. This amount varies from 400 to 700 feet, depending on own aircraft's altitude.* ALIM *includes allowances to account for intruder and own altimetry errors and vertical tracking uncertainties that affect track projections (see PR22.3). The value of* ALIM *increases with altitude to reflect increased altimetry error (↑ SC4.5) and the need to increase tracked separation at higher altitudes.*

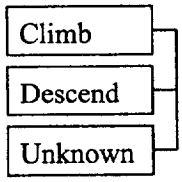
The blackbox behavioral specification shown in Figure 7 also points to the module that implements this required behavior in the design specification on Level 4. For TCAS II, pseudocode was used for the design specification. Figure 8 shows the pseudocode provided by MITRE for the Don't-Care-Test.

The structure of intent specifications has advantages in solving various software engineering problems—such as changing requirements, program understanding, maintaining and changing code, and validation—as discussed in the next section.

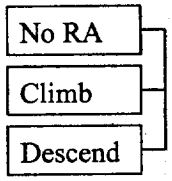
AIRCRAFT(i).STATUS



AIRCRAFT(i).SENSE



OWN-RA-SENSE



Macro: Don't-Care-Test(i)

		OR			
	Aircraft(i).Capability $_{v\text{-}392}$ = TCAS-TA/RA	F	F	F	F
	Aircraft(i).Status **in-state** New	T	T	T	T
A	Aircraft(i).Sense **in-state** Climb	T	.	T	.
N	Aircraft(i).Sense **in-state** Descend	.	T	.	T
D	Down-Separation $_{f\text{-}517}$ < ALIM [⋀PR1]	F	.	.	.
	Up-Separation $_{f\text{-}542}$ < ALIM [⋀PR1]	.	F	.	.
	Own-RA-Sense **in-state** Descend	.	.	T	.
	Own-RA-Sense **in-state** Climb	.	.	.	T
	Some Aircraft(j).Sense **not-in-same-state-as** Aircraft(i).Sense	.	.	T	T
	Some Aircraft(j).Vertical-Miss-Distance $_{f\text{-}543}$ (RELALT, TAUM, TRTRU, TVPE) < Separation-Second-Choice(i) $_{f\text{-}538}$	.	.	T	T

Comment: The last two entries in the AND/OR table ensure that there exists at least one other aircraft that is a threat and has selected a sense opposite that of the current aircraft, and that the modeled separation for that aircraft following a leveloff is worse than the modeled separation for the current aircraft in the opposite (second choice) sense.

Mapping to Level 2: ⋀PR35

Mapping to Level 4: ⋁Sense.Dont-care-test

Abbreviations:

ALIM = Positive-RA-Altitude-Limit-Threshold $_{t\text{-}545}$ [Alt-Layer-Value $_{f\text{-}510}$]

RELALT = Own-Tracked-Alt $_{f\text{-}529}$ + (4 s x Own-Tracked-Alt-Rate $_{f\text{-}528}$) - Other-Tracked-Alt $_{f\text{-}524}$

TAUM = Min (Max (Modified-Tau-Capped $_{f\text{-}522}$, 10 s, True-Tau-Uncapped $_{f\text{-}542}$)

TRTRU = True-tau-Capped $_{f\text{-}542}$

TVPE = XTVPETBLX $_{t\text{-}552}$ [Other-Sensitivity-Level $_{v\text{-}391}$]

Figure 7: This macro is used in the defining which resolution advisory will be chosen when multiple aircraft (threats) are involved, among the most complicated aspects of the collision avoidance logic. Abbreviations are used to enhance readability.

```
PROCESS Sense.Don't_care_test;

        (↑)Don't_Care_Test_{m-357}, (↑)Climb_Desc._Inhibit_{m-317}

          {WL threat = threat whose WL entry is input to task}
          {TF threat = threat examined in loop below}
          IF (either sense provides adequate separation)
             THEN SET Don't_care flag for WL threat;
             ELSE CLEAR Don't_care flag for WL threat;
                  IF (own resolution advisories show a Positive in second-choice sense)
                     THEN calculate own altitude following a leveloff;
                          REPEAT WHILE (more entries in threat file AND don't_care flag
                                  for WL threat not set);
                                  IF (resolution against TF threat shows a Positive in same sense
                                     as second choice for WL threat)
                                     THEN calculate altitude relative to TF threat and
                                             time for leveloff;
                                             {result of 'do care' for WL threat}
                                             CALL vertical_miss_distance_calculation
                                                     IN (rel alt, rel vert rate, start time (WL threat)
                                                        end time (WL threat), clip time (WL threat));
                                             IF (sep with leveloff vs. TR threat less than that
                                                     for second choice maneuver vs. WL threat)
                                                        THEN SET Don't_care flag for the WL threat;
                                                                {allow second choice sense}
                                  Select next threat file entry;
                          ENDREPEAT;
END Don't_care_test;
```

Figure 8: The pseudocode for the Don't-Care-Test.

5 Intent Specification Support for Software Engineering Problem Solving

As stated earlier, our representations of problems have an important effect on our problem-solving ability and the strategies we use. A basic hypothesis of this paper is that intent specifications will support the problem solving required to perform software engineering tasks. This hypothesis seems particularly relevant with respect to tasks involving education and program understanding, search, design, validation, safety assurance, maintenance, and evolution.

5.1 Education and Program Understanding

Curtis *et.al.* [CKI88] did a field study of the requirements and design process for 17 large systems. They found that substantial design effort in projects was spent coordinating a common understanding among the staff of both the application domain and of how the system should perform within it. The most successful designers understood the application domain and were adept at identifying unstated requirements, constraints, or exception conditions and mapping between these and the computational structures. This is exactly the information that is included in the higher levels of intent specifications and the mappings to the software. Thus using intent specifications should help with education in the most crucial aspects of the system design for both developers and maintainers and augment the abilities of both, i.e., increase the intellectual manageability of the task.

5.2 Search Strategies

Vicente and Rasmussen have noted that means-ends hierarchies constrain search in a useful way by providing traceability from the highest level goal statements down to implementations of the components [VR92]. By starting the search at a high level of abstraction and then deciding which part of the system is relevant to the current goals, the user can concentrate on the subtree of the hierarchy connected to the goal of interest: The parts of the system not pertinent to the function of interest can easily be ignored. This type of "zooming-in" behavior has been observed in a large number of psychological studies of expert problem solvers. Recent research on problem-solving behavior consistently shows that experts spend a great deal of their time analyzing the functional structure of a problem at a high level of abstraction before narrowing in on more concrete details [BP87, BS91, GC88, Ras86, Ves85].

With other hierarchies, the links between levels are not necessarily related to goals. So although it is possible to use higher levels of abstraction in a standard decomposition or refinement hierarchy to select a subsystem of interest and to constrain search, the subtree of the hierarchy connected to a particular subsystem does not necessarily contain system components that are relevant to the goals and constraints that the problem solver is considering.

Upward search in the hierarchy, such as that required for debugging, is also supported by intent specifications. Vicente and Rasmussen claim (and have experimental evidence to support) that in order for operators to correctly and consistently diagnose faults, they must have access to higher-order functional information since this information provides a reference point defining how the system ***should*** be operating. States can only be described as errors or faults with reference to the intended purpose. Additionally, causes of improper functioning depend upon aspects of the implementation. Thus they are explained bottom up. The same argument seems to apply to software debugging. There is evidence to support this hypothesis. Using protocol analysis, Vessey found that the most successful debuggers had a "system" view of the software [Ves85].

5.2.1 Design Criteria and Evaluation

An interesting implication of intent specifications is their potential effect on system and software design. Such specifications might not only be used to understand and validate designs but also to guide them.

An example of a design criterion appropriate to intent specifications might be to minimize the number of one-to-many mappings between levels in order to constrain downward search and limit the effects of changes in higher levels upon the lower levels. Minimizing many-to-many (or many-to-one) mappings would, in addition, ease activities that require following upward links and minimize the side effects of lower-level changes.

Intent specifications assist in identifying intent-related structural dependencies (many-to-many mappings across hierarchical levels) to allow minimizing them during design, and they clarify the tradeoffs being made between conflicting goals. Software engineering attempts to define coupling between modules have been limited primarily to the design level. Perhaps an intent specification can provide a usable definition of coupling with respect to emergent properties and to assist in making design tradeoffs between various types of high-level coupling.

5.3 Minimizing the Effects of Requirements Changes

Hopefully, the highest levels of the specification will not change, but sometimes they do, especially during development as system requirements become better understood. Functional and intent aspects are represented throughout an intent specification, but in increasingly abstract and global terms at the higher levels. The highest levels represent more stable design goals that are less likely to change (such as detecting potential threats in TCAS), but when they do they have the most important (and costly) repercussions on the system and software design and development, and they may require analysis and changes at all the lower levels. We need to be able to determine the potential effects of changes and, proactively, to design to minimize them.

Reversals in TCAS are an example of this. About four years after the original TCAS specification was written, experts discovered that it did not adequately cover requirements involving the case where the pilot of an intruder aircraft does not follow his or her TCAS advisory and thus TCAS must change the advisory to its own pilot. This change in basic requirements caused extensive changes in the TCAS design, some of which introduced additional subtle problems and errors that took years to discover and rectify.

Anticipating exactly what changes will occur and designing to minimize the effects of those changes is difficult, and the penalties for being wrong are great. Intent specifications theoretically provide the flexibility and information necessary to design to ease high-level requirements changes without having to predict exactly which changes will occur: The abstraction and design are based on intent (system requirements) rather than on part-whole relationships (which are the least likely to change with respect to requirement or environment changes).

5.4 Design of Run-Time Assertions

Finally, intent specifications may assist software engineers in designing effective fault-tolerance mechanisms. Detecting unanticipated faults during execution has turned out to be a very difficult problem. For example, in one of our empirical studies, we found that programmers had difficulty writing effective assertions for detecting errors in executing software [LCKS90]. I have suggested that using results from safety analyses might help in determining which assertions are required and where to detect the most important errors [Lev91]. The information in intent specifications tracing intent from requirements, design constraints, and hazard analyses through the system and software design process to the software module (and back) might assist with writing effective and useful assertions to detect general violations of system goals and constraints.

5.5 Safety Assurance

A complete safety analysis and methodology for building safety-critical systems requires identifying the system-level safety requirements and constraints and then tracing them down to the components [Lev95]. After the safety-critical behavior of each component has been determined (including the implications of its behavior when the components interact with each other), verification is required that the components do not violate the identified safety-related behavioral constraints. In addition, whenever any change is made to the system or when new information is obtained that brings the safety of the design into doubt, revalidation is required to ensure that the change does not degrade system safety. To make this verification (and reverification) easier, safety-critical parts of the software should be isolated and minimized.

This analysis cannot be performed efficiently unless those making decisions about changes and those actually making the changes know which parts of the system affect a particular safety design constraint. Specifications need to include a record of the design decisions related to basic safety-related system goals, constraints, and hazards (including both general design principles and criteria and detailed design decisions), the assumptions underlying these decisions, and why the decisions were made and particular design features included. Intent specifications capture this information and provide the ability to trace design features upward to specific high-level system goals and constraints.

5.6 Software Maintenance and Evolution

Although intent specifications provide support for a top-down, rational design process, they may be even more important for the maintenance and evolution process than for the original designer, especially of smaller or less complex systems. Software evolution is challenging because it involves many complex cognitive processes—such as understanding the system's structure and function, understanding the code and documentation and the mapping between the two, and locating inconsistencies and errors—that require complex problem-solving strategies.

Intent specifications provide the structure required for

recording the most important design rationale information, i.e., that related to the purpose and intent of the system, and locating it when needed. They, therefore, can assist in the software change process.

While trying to build a model of TCAS, we discovered that the original conceptual model of the TCAS system design had degraded over the years as changes were made to the pseudocode to respond to errors found, new requirements, better understanding of the problem being solved, enhancements of various kinds, and errors introduced during previous changes. The specific changes made often simplified the process of making the change or minimized the amount of code that needed to be changed, but complicated or degraded the original model. Not having any clear representation of the model also contributed to its degradation over the ten years of changes to the pseudocode.

By the time we tried to build a representation of the underlying conceptual model, we found that the system design was unnecessarily complex and lacked conceptual coherency in many respects, but we had to match what was actually flying on aircraft. I believe that making changes without introducing errors or unnecessarily complicating the resulting conceptual model would have been simplified if the TCAS staff had had a blackbox requirements specification of the system. Evolution of the pseudocode would have been enhanced even more if the extra intent information had been specified or organized in a way that it could easily be found and traced to the code.

Tools for restructuring code have been developed to cope with this common problem of increasing complexity and decreasing coherency of maintained code [GN95]. Using intent specifications will not eliminate this need, but we hope it will be reduced by providing specifications that assist in the evolution process and, more important, assist in building software that is more easily evolved and maintained. Such specifications may allow for backing up and making changes in a way that will not degrade the underlying conceptual model because the model is explicitly described and its implications traced from level to level. Intent specifications may also allow controlled changes to the higher levels of the model if they become necessary.

Maintenance and evolution research has focused on ways to identify and capture information from legacy code. While useful for solving important short-term problems, our long term goal should be to specify and design systems that lend themselves to change easily—that is, evolvable systems. Achieving this goal requires devising methodologies that support change throughout the entire system life cycle—from requirements and specification to design, implementation and maintenance. For example, we may be able to organize code in a way that will minimize the amount of code that needs to be changed or that needs to be evaluated when deciding if a change is safe or reasonable.

In summary, I believe that effective support for such evolvable systems will require a new paradigm for specification and design and hypothesize that such a paradigm might be rooted in abstractions based on intent. Intent specifications provide the framework to include the information maintainers need in the specification. They increase the information content so that less inferencing (and guessing) is required. Intent specifications not only support evolution and maintenance, but they may be more evolvable themselves, which would ease the problem of keeping documentation and implementation consistent. In addition, they also provide the possibility of designing for evolution so that the systems we build are more easily maintained and evolved.

6 Conclusions

Specifications are constructed to help us solve problems. Any theory of specification design, then, should be based on fundamental concepts of problem-solving behavior. It should also support the basic systems engineering process. This paper has presented one such approach to system and software specifications based on underlying ideas from psychology, systems theory, human factors, system engineering, and cognitive engineering.

The choice of content, structure, and form of specifications have a profound effect on the kind of cognitive processing that the user must bring to bear to use a specification for the tasks involved in system and software design and construction, maintenance, and evolution. Intent specifications provide a way of coping with the complexity of the cognitive demands on the builders and maintainers of automated systems by basing our specifications on means–ends as well as part–whole abstractions. I believe that the levels of the means-ends hierarchy reflect a rational design philosophy for the systems engineering of complex systems and thus a rational way to specify the results of the process. They provide mapping (tracing) of decisions made earlier into the later stages of the process. Design decisions at each level are linked to the goals and constraints they are derived to satisfy. A seamless (gapless) progression is recorded from high-level system requirements down to component requirements, design, and implementation.

In addition, intent specifications provide a way of integrating formal and informal aspects of specifications. Completely informal specifications of complex systems

tend to be unwieldy and difficult to validate. Completely formal specifications provide the potential for mathematical analysis and proofs but omit necessary information that cannot be specified formally. Some formal approaches require building special models in addition to the regular system specifications. I believe that the wide-spread use of formal specifications in industry will require the development of formal specifications that are readable with minimal training requirements and that are integrated with informal specifications. Ideally, formal analysis should not require building special models that duplicate the information included in the specification or it is unlikely that industry will find the use of formal methods to be cost effective.

An example intent specification for TCAS II has been constructed and was used as an example in this paper. The reader is cautioned, however, that intent specifications are a logical abstraction that can be realized in many different physical ways. That is, the particular organization used for the TCAS specification is simply one possible physical realization of the general logical organization inherent in intent specifications.

7 REFERENCES

[AT90] D. Ackermann and M. J. Tauber, editors. *Mental Models and Human-Computer Interaction.* North-Holland, Amsterdam, 1990.

[Ash62] W.R. Ashby. Principles of the self-organizing system. in H. Von Foerster and G.W. Zopf (eds.) *Principles of Self-Organization*, Pergamon, 1962.

[BP87] M. Beveridge and E. Parkins. Visual representation in analogical program solving. *Memory and Cognition*, v. 15, 1987.

[Bro83] R. Brooks. Towards a theory of comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.

[BL98] M. Brown and N. G. Leveson. Modeling Controller Tasks for Safety Analysis. *Second Workshop on Human Error and System Development*, Seattle, April 1998.

[BS91] M.A. Buttigieg and P.M. Sanderson. Emergent features in visual display design for two types of failure detection tasks. *Human Factors*, 33, 1991.

[Cas91] S.M. Casner. A task analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, vol. 10, no. 2, April 1991.

[Che81] P. Checkland. *Systems Thinking, Systems Practice.* John Wiley & Sons, 1981.

[CKI88] B. Curtis, H. Krasner and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(2): 1268–1287, 1988.

[DB83] DeKleer J, and J.S. Brown. Assumptions and ambiguities in mechanistic mental models. In D. Gentner and A.L. Stevens (eds.), *Mental Models*, Lawrence Erlbaum, 1983.

[DV96] N. Dinadis and K.J. Vicente. Ecological interface design for a power plant feedwater subsystem. *IEEE Transactions on Nuclear Science*, in press.

[Dor87] D. Dorner. On the difficulties people have in dealing with complexity. In Jens Rasmussen, Keith Duncan, and Jacques Leplat, editors, *New Technology and Human Error*, pages 97-109, John Wiley & Sons, New York, 1987.

[Dun87] K.D. Duncan. Reflections on fault diagnostic expertise. In Jens Rasmussen, Keith Duncan, and Jacques Leplat, editors, *New Technology and Human Error*, pages 261-269, John Wiley & Sons, New York, 1987.

[FSL78] B. Fischoff, P. Slovic, and S. Lichtenstein. Fault trees: Sensitivity of estimated failure probabilities to problem representation. *Journal of Experimental Psychology: Human Perception and Performance*, vol. 4, 1978.

[FG79] Fitter and Green. When do diagrams make good programming languages?. *Int. J. of Man-Machine Studies*, 11:235–261, 1979.

[GC88] R. Glaser and M. T. H. Chi. Overview. In R. Glaser, M. T. H. Chi, and M. J. Farr, editors, *The Nature of Expertise.* Erlbaum, Hillsdale, New Jersey, 1988.

[GN95] W. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, March 1995.

[Har82] G. Harman. Logic, reasoning, and logic form. In *Language, Mind, and Brain*, T.W. Simon and R.J. Scholes (eds.), Lawrence Erlbaum Associates, 1982.

[JLHM91] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B.Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. on Software Engineering*, SE-17(3), March 1991.

[KS90] C.A. Kaplan and H.A. Simon. In search of insight. *Cognitive Psychology*, vol. 22, 1990.

[KHS85] K. Kotovsky, J.R. Hayes, and H.A. Simon. Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology*, vol. 17, 1985.

[Let86] S. Letovsky. Cognitive processes in program comprehension. In *Proceedings of the First Workshop on Empirical Studies of Programmers*, pages 58–79. Ablex Publishing, Norwood, NJ, 1986.

[Lev91] N.G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, vol. 34, no. 2, February 1991.

[Lev95] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, 1995.

[LCKS90] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self-checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, vol. SE-16, no. 4, April 1990.

[LHHR94] N.G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *Trans. on Software Engineering*, SE-20(9), September 1994.

[LPS97] N.G. Leveson, L.D. Pinnel, S.D. Sandys, S. Koga, and J.D. Reese. Analyzing software specifications for mode confusion potential. Workshop on Human Error and System Development, Glascow, March 1977.

[Luc87] D.A. Lucas. Mental models and new technology. In Jens Rasmussen, Keith Duncan, and Jacques Leplat, editors, *New Technology and Human Error*, pages 321-325. John Wiley & Sons, New York, 1987.

[New66] J.R. Newman. Extension of human capability through information processing and display systems. Technical Report SP-2560, System Development Corporation, 1966.

[Nor93] D.A. Norman. *Things that Make us Smart.* Addison-Wesley Publishing Company, 1993.

[RP95] J. Rasmussen and A. Pejtersen. Virtual ecology of work. In J. M. Flach, P. A. Hancock, K. Caird and K. J. Vicente, editors *An Ecological Approach to Human Machine Systems I: A Global Perspective*, Erlbaum, Hillsdale, New Jersey, 1995..

[Pen87] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19:295-341,1987.

[Ras85] J. Rasmussen. The Role of hierarchical knowledge representation in decision making and system management. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, no. 2, March/April 1985.

[Ras86] J. Rasmussen. *Information Processing and Human–Machine Interaction: An Approach to Cognitive Engineering.* North Holland, 1986.

[Ras90] J. Rasmussen. Mental models and the control of action in complex environments. In D. Ackermann and M.J. Tauber (eds.) *Mental Models and Human–Computer Interaction*, Elsevier (North-Holland), 1990, pp. 41–69.

[Rea90] J. Reason. *Human Error.* Cambridge University Press, 1990.

[SWB95] N.D. Sarter, D.D. Woods, and C.E. Billings. Automation Surprises. in G. Salvendy (Ed.) *Handbook of Human Factors/Ergonomics*, 2nd Edition, Wiley, New York, in press.

[SM79] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Computer and Info. Sciences*, 8(3):219–238, 1979.

[Smi89] G.F. Smith. Representational effects on the solving of an unstructured decision problem. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-19, 1989, pp. 1083-1090.

[SE84] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, vol. SE-10(5):595–609, 1984.

[Sol88] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(2): 1259–1267, 1988.

[Ves85] I. Vessey. Expertise in debugging computer programs: A process analysis. *Int. J. of Man–Machine Studies*, vol. 23, 1985.

[Vic91] K.J. Vicente. Supporting knowledge-based behavior through ecological interface design. Ph.D. Dissertation, University of Illinois at Urbana-Champagne, 1991.

[VCP95] K.J. Vicente, K. Christoffersen and A. Pereklit. Supporting operator problem solving through ecological interface design. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(4):529–545, 1995.

[VR90] K.J. Vicente and J. Rasmussen. The ecology of human–machine systems II: Mediating direct perception in complex work domains. *Ecological Psychology*, 2(3):207–249, 1990.

[VR92] K.J. Vicente and J. Rasmussen. Ecological interface design: Theoretical foundations. *IEEE Trans. on Systems, Man, and Cybernetics*, vol 22, No. 4, July/August 1992.

[Wei89] E.L. Wiener. *Human Factors of Advanced Technology ("Glass Cockpit") Transport Aircraft.* NASA Contractor Report 177528, NASA Ames Research Center, June 1989.

[Woo95] D.D. Woods. Toward a theoretical base for representation design in the computer medium: Ecological perception and aiding human cognition. In J. M. Flach, P. A. Hancock, K. Caird and K. J. Vicente, editors *An Ecological Approach to Human Machine Systems I: A Global Perspective*, Erlbaum, Hillsdale, New Jersey, 1995.